Sistemas Informáticos
Curso 2004-2005

GENARO

Javier Gómez Santos
Juan Rodríguez Hortalá
Roberto Torres de Alba

Dirigido por:
Prof. Jaime Sánchez Hernández
Dpto. Sistemas Informáticos y Programación

Facultad de Informática
Universidad Complutense de Madrid

Resumen:
Este proyecto consiste en el desarrollo de una herramienta de ayuda a la composición, que permita generar archivos midi correspondiente a piezas de varias secciones y para varios instrumentos. Tendrá dos usos principales, como generador de nuevas ideas musicales a través de algoritmos con una componente aleatoria, y como ayudante en la composición, encargándose de tareas repetitivas en la construcción de un midi. El proceso de composición será interactivo a través de un interfaz de usuario con ventanas.

Abstract:
This project consists of the development of a tool of aid to the composition, that allows to generate archives midi corresponding to pieces of several sections and for several instruments. It would have two main uses, like generator of new musical ideas through algorithms with a random component, and like assistant in the composition, being in charge of repetitive tasks in construccon of midi. The composition process would be interactive through an user interface with windows.

música, composicíon, aleatorio, creatividad, melodía, ritmo, acento, Genaro, armonía, jazz

Índice general

Capítulo 1

Introducción a la música

1.1. ¿Qué es una nota?

Para intentar formalizar la música consideremos el mínimo elemento del que se compone cualquier creación musical, una nota. Una *nota* es un sonido al que se le han aplicado dos restricciones:

- De duración: se espera que el sonido que es la nota permanezca sonando durante una cantidad de tiempo determinada. Esta cantidad de tiempo se especifica mediante una *figura*, que no es sino una fracción que expresa la duración de la nota en relación con una medida de tiempo absoluta llamada *tempo*, que se especifica para toda la obra musical o para cada fragmento de ésta.
- De altura: se espera que el sonido tenga una altura determinada en la escala de medida musical, es decir, que sea más o menos grave o agudo. Las *alturas musicales* no son sino frecuencias de oscilación de ondas sonoras, se consideran ciertas frecuencias como válidas y entonces se dice que una nota está *afinada* cuando corresponde a una de estas frecuencias. Por tanto la altura de una nota es una correspondencia con una de estas frecuencias afinadas. Existen varios sistemas de afinación (justa, temperada, ...) que no son sino correspondencias entre nombres de alturas musicales y frecuencias de oscilación.

Por último hay que tener en cuenta que en la música la ausencia de sonidos es tan importante como su presencia. Por ello definimos un tipo de notas especiales que son los *silencios*, que son notas con una duración asociada pero sin una frecuencia asociada (o también podemos considerar que están asociadas a una frecuencia especial). Los silencios en una composición para un instrumento corresponden con los momentos en que dicho instrumento no emite ningún sonido.

1.2. Dimensiones de la música

Habiendo entendido lo que es una nota podemos intentar desarrollar una visión más amplia de la música. Si entendemos las dos restricciones que se imponen sobre las notas como restricciones sobre dos dimensiones sobre las que se desarrolla la musica.

- Horizontal/Temporal: la música discurre a lo largo del tiempo.

- Vertical/Altura: los componentes de la música, las notas, tienen unas alturas definidas, es decir, se disponen a lo largo de un eje vertical.

Por tanto la música se puede representar como una agrupación de 'puntos' (las notas), dispuestas en el espacio según dos ejes: el horizontal o temporal, que determina qué sonidos suenan o dejan de sonar, antes o después; y el vertical o de alturas, que determina la altura de los sonidos en el momento de su instanciación.

1.3. Otros conceptos básicos

En esta sección introduciremos algunos conceptos que necesitamos manejar para entender el funcionamiento de GENARO

Acorde: El concepto de *acorde* se puede entender a distintos niveles de abstracción. En el nivel más concreto, un acorde es un grupo de dos o más notas que suenan a la vez. Pero a un nivel más alto de abstracción un acorde se puede entender como una agrupación de alturas musicales que se considera que tienen un significado conjunto, y llamaremos *cifrado* a la representación de los acordes a este nivel de abstracción.

Tonalidad: Es un eje o contexto global que define la estabilidad de las notas musicales y la forma en la que se suceden los acordes. GENARO trabaja siempre en tonalidades mayores (en realidad trabaja siempre en Do Mayor y luego traspone el resultado).

Compás: Es una unidad de tiempo en la que se divide una frase u obra musical, cada compás está dividido en periodos de tiempo de igual duración llamados "tiempos". Hay varios tipos de compases, GENARO trabaja siempre en el compás binario de 2/2. Este tipo de compás se caracteriza por ser la sucesión de un tiempo fuerte y un tiempo débil.

Escala: Un acorde define una jerarquía de sonidos en el intervalo de tiempo en el que este está vigente. Una escala es un conjunto de alturas musicales que se "permiten" durante la duración del acorde. Dentro de las notas de las alturas que pertenecen a la escala, algunas son más estables y otras menos.

1.4. Enfoque GENARO de la música

Se puede entender que la música esta formada por tres elementos, *melodía*, *armonía* y *ritmo*:

- Melodía: se refiere a la sucesión de notas a lo largo del tiempo. La melodía de una composición es lo que la gente suele cantar e identificar más claramente.
- Armonía: se refiere a la música considerada según su eje vertical, es decir, a las relaciones de altura entre distintas notas que suenan a la vez. A estos conjuntos de notas simultáneas los llamamos *acordes*.
- Ritmo: se refiere a la repetición de patrones de duraciones de notas a lo largo del eje temporal.

GENARO se inspira en una de las formaciones clásicas del Jazz, el trío base, a la hora de enfocar la composición de música. Un trío base está compuesto por un piano, un contrabajo y una batería. Estos instrumentos tienen unas funciones o papeles bien definidos, para poder dar cobertura a los tres elementos de los que se compone la música:

- *Batería*: se ocupa de dar soporte al ritmo.
- *Bajo*: se ocupa de dar apoyo rítmico a la bateria y sobre todo de sustentar la armonía junto con el piano. Ocasionalmente puede realizar funciones melódicas en mayor o menor medida.
- Piano: el piano se puede entender como dos instrumentos en uno porque cada mano del piano es independiente (si el pianista es bueno). Debido a ello en GENARO consideramos por separado cada mano del piano:
  (a) Mano izquierda: esta mano se ocupa de apoyar al ritmo y sobre todo de la armonía. En GENARO llamamos *acompañamiento* a este 'instrumento'.
  (b) Mano derecha: la mano derecha del piano lleva casi todo el peso melódico. En GENARO llamamos *melodía* a este 'instrumento'.

En GENARO llamamos *tipo de pista* a estos 'instrumentos' y la música se genera de forma diferente para cada tipo de pista, adecuándose a sus características. Al final del proyecto no dio tiempo a terminar el compositor para pistas de batería, pero si quedaron listos los compositores de acompañamiento, bajo y melodía.

Capítulo 2

Manual de GENARO

2.1. Introducción

2.1.1. ¿Qué es GENARO?

GENARO es una herramienta de ayuda a la composición, una opción a la que recurrir si se quiere buscar ideas nuevas. Para tal fin, GENARO compone, mediante ciertos parámetros especificados por el usuario, una pieza sencilla. Como esto es un proceso bastante rápido, el usuario puede en un pequeño espacio de tiempo disponer de varias ideas para poder buscar en ellas alguna inspiración.

2.1.2. Historia de GENARO

GENARO es un programa creado por Javier Gómez Santos, Juan Rodríguez Hortalá y Roberto Torres de Alba, estudiantes de Ingeniería Informática de la Universidad Complutense de Madrid. Surgió como un proyecto para la universidad, en el que no se tenía muy claro hasta donde podría llegarse. Algunos de las propuestas iniciales no pudieron ser alcanzadas, pero muchas de ellas si. Ahora GENARO es un software gratuito para ser usado por cualquiera que quiera buscar ayuda a la composición, o que le apetezca sencillamente jugar un poco con el programa.

2.1.3. ¿Qué es lo que hace GENARO?

GENARO compone fragmentos musicales para los distintos tipos de pista basándose en los fragmentos anteriormente compuestos para otras pistas. De esta manera los fragmentos musicales que suenan a la vez están relacionados y 'suenan bien' al reproducirse simultáneamente. En la versión actual de GENARO hay dos maneras de generar música:

1. El acompañamiento manda: se genera primero el acompañamiento y a partir de él se genera la melodía y el bajo.

2. Armonizador: a partir de una melodía se genera un acompañamiento a partir del cuál se puede generar el bajo.

GENARO permite introducir en sus proyectos cualquier número de pistas de un tipo a elegir entre los tres desarrollados (acompañamiento, bajo y melodía), pudiendo tener cualquier número de pistas

del mismo tipo. Las pistas se corresponden con instrumentos que pertenecen a uno de los tres tipos desarrollados.

El producto último que nos proporciona GENARO son archivos midi o wav:

- Midi: es un estándar en la codificación de partituras por ordenador, permitiendo por tanto al usuario editar y manipular la música compuesta por GENARO con cualquiera de los secuenciadores y programas de edición del mercado. Representa la música compuesta como partitura, sin fijar del todo los instrumentos sintéticos que interpretaran cada pista. Dichos instrumentos se fijarán definitivamente en el programa que haga de sintetizador del fichero.
- Wav: es un estándar en codificación de audio. GENARO tiene incorporado un sintetizador software, Timidity++ (vease sección 11.1 de la página 83), que puede realizar el paso a wav del midi generado, asociando a cada pista una fuente de sonido seleccionado de una extensa lista.

Debido a las múltiples pistas y a la posibilidad de asignar fuentes de sonidos de muchos tipos a cada pista, GENARO es capaz de generar músicas muy diversas. En su forma de uso más básica se puede generar un acompañamiento de piano y una melodia de piano con un bajo acústico de fondo. Pero pensando un poco surgen más opciones, por ejemplo se puede añadir otra pista de acompañamiento que doble a la anterior pero con una fuente de sonido que sean una sección de cuerda; o añadir más pistas de melodía que se contrapongan a la primera, usando fuentes de sonido distintas; o silenciar todo menos el bajo y añadir más pistas de bajo, con lo que se obtiene una polifonía al estilo barroco; o introducir otra pista de acompañamiento de piano pero con otro ritmo, para obtener ritmos compuestos...

2.1.4. ¿Cómo funciona?

GENARO ha sido elaborado en tres lenguajes distintos, Prolog, Haskell y C++. En los módulos desarrollados en prolog y haskell es donde residen los algoritmos de inteligencia artificial, que son el núcleo de GENARO, donde se desarrolla la generación de música. C++ por su parte se encarga de crear un interfaz gráfico agradable para el usuario, y actúa como nexo entre los demás lenguajes.

2.2. Instalación

GENARO no dispone actualmente de instalador, si te has descargado GENARO de internet, lo más probable es que esté comprimido en un fichero zip o similar. Tan sólo descomprímelo al directorio que desees, y ejecuta el programa *InterfazGenaro.exe* que se halla en el subdirectorio *Codigo/C*.

2.3. Manejando a GENARO

2.3.1. Arrancando el programa

GENARO posee varios ejecutables, pero son casi todos auxiliares, y no deberían usarse si no se posee un conocimiento claro de como van a reaccionar. Es aconsejable pues ejecutar el programa principal, de nombre *InterfazGenaro.exe*, que se halla en el subdirectorio *Codigo/C*. Este programa se encarga de unir a todos los restantes mediante llamadas totalmente transparentes para el usuario. Una llamada al programa desde otro directorio al suyo, podra causar problemas para conseguir encontrar los demás programas y archivos necesarios. Una vez arrancado el interfaz principal, una nueva ventana debería

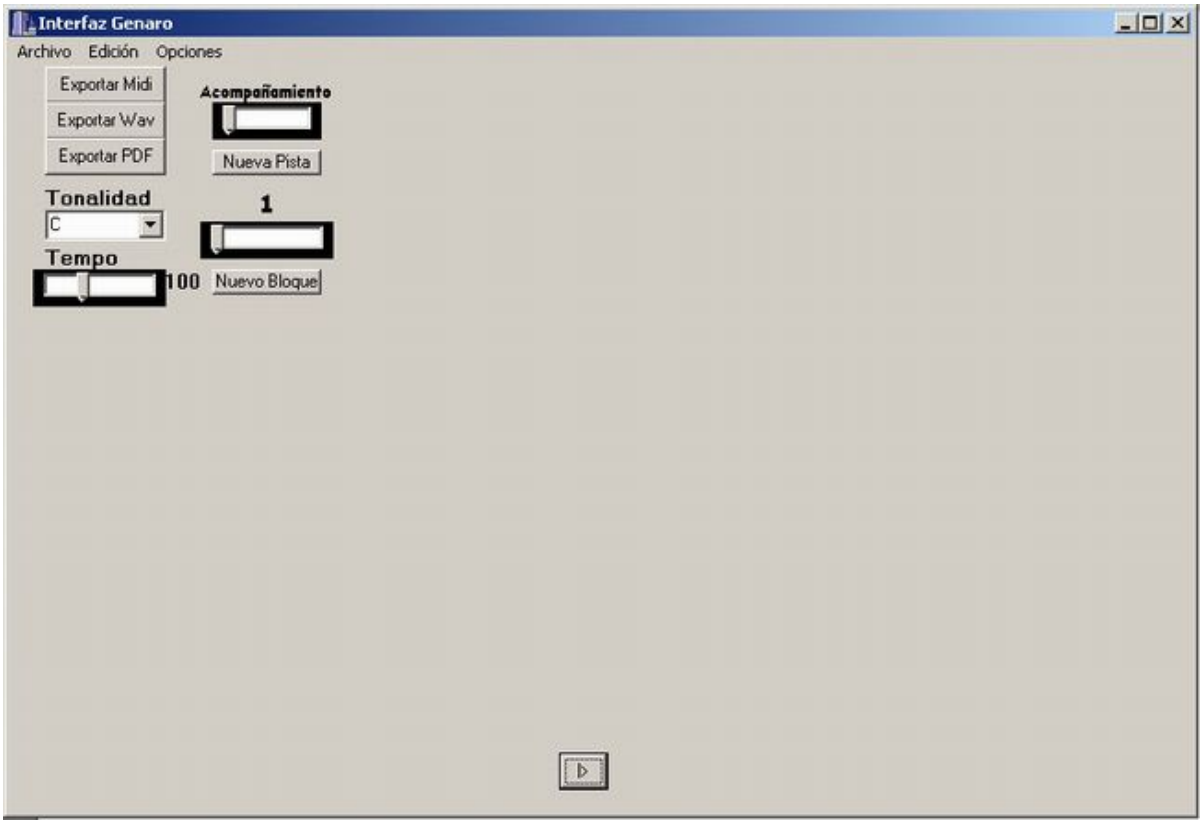


Figura 2.1: Interfaz vacío tras arrancar

aparecer, similar a la mostrada en la figura 2.1 de la página 14, que ha sido modificada con el fin de hacerla más fácil de ver.

2.3.2. Creando un nuevo proyecto

Una vez hemos arrancado el programa, hemos de crear un nuevo proyecto. Para ello debemos dirigirnos al menú *Archivo*, y escoger la opción *Nuevo*, tal y como se muestra en la figura 2.2 de la página 15. Tras esto nos debería quedar algo parecido a lo mostrado en la figura 2.3 de la página 16.

2.3.3. Creando pistas

Un proyecto GENARO consta de un número variable de pistas. Una pista es un conjunto de sonidos asignados a un instrumento. Para GENARO existen 3 tipos distintos de pistas, la pista de *acompañamiento*, la pista de *melodía* y la pista de *bajo*. Cada una de estas pistas funciona de manera distinta, y puede haber cualquier número de ellas, a excepción de la pista de acompañamiento, ya que ha de haber siempre un mínimo de 1.
Para crear una nueva pista, debemos mover el selector que se encuentra situado encima del botón *Nueva*

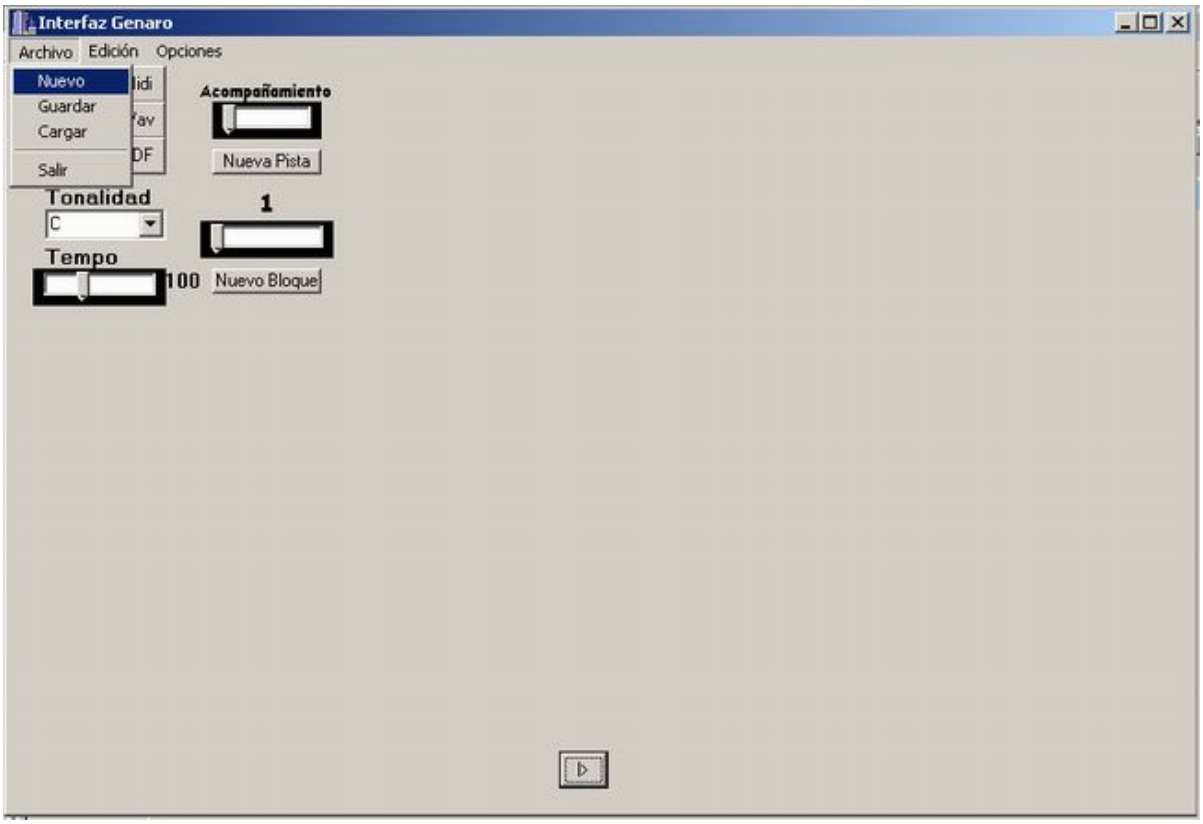


Figura 2.2: Creando un nuevo proyecto

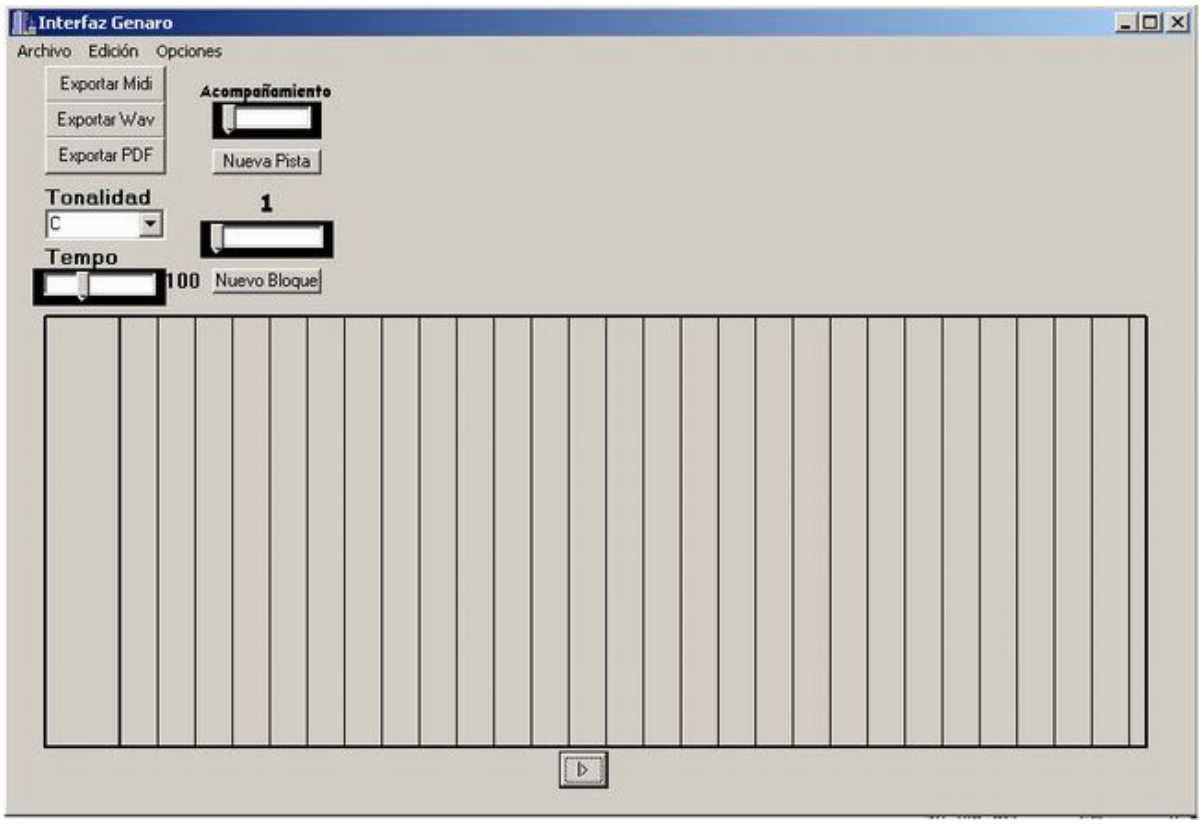


Figura 2.3: Interfaz tras pulsar el botón *nuevo*

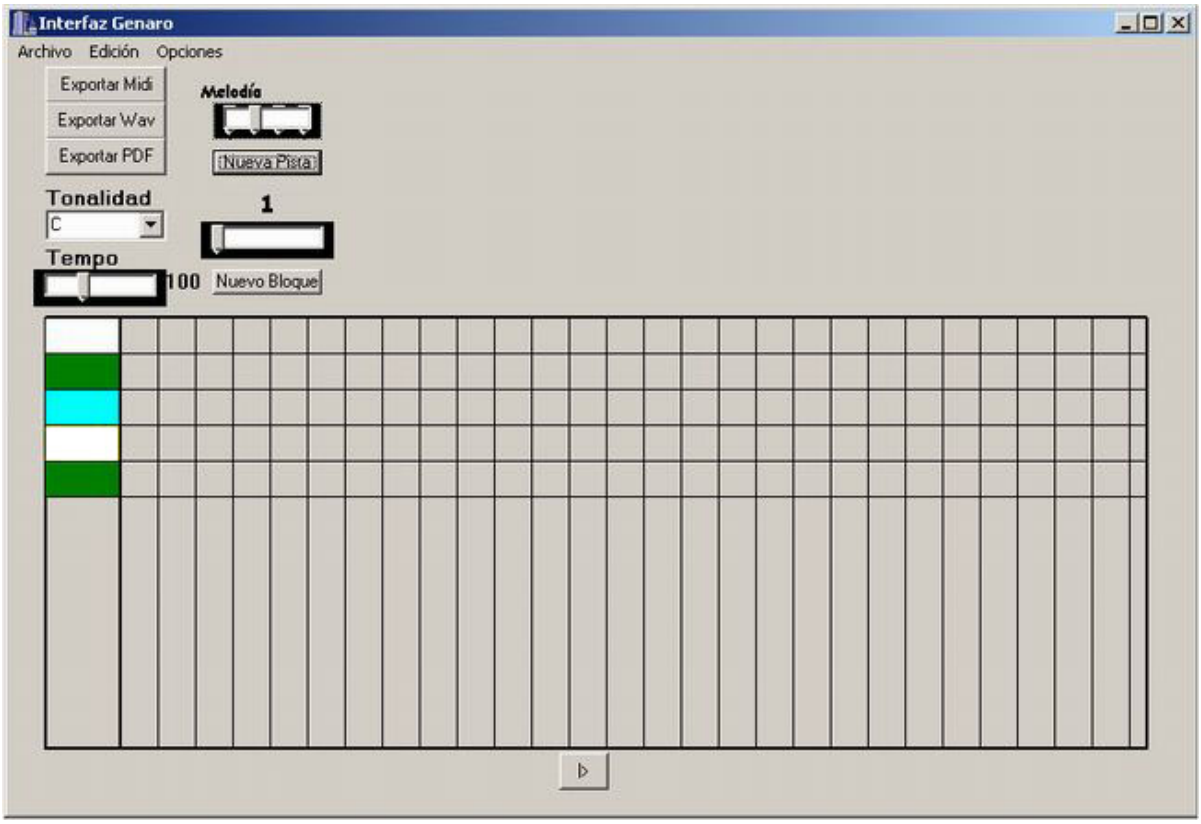


Figura 2.4: Creando varias pistas

Pista, arriba a la izquierda. Al desplazarlo vemos como cambia el tipo de pista seleccionado, mostrando cualquiera de los anteriormente mencionados. Cuando hayamos encontrado el que buscamos, pulsamos el botón *Nueva Pista*. Una nueva división vertical debe haber aparecido, con un cuadro coloreado a la izquierda. Esto es una pista. En la figura 2.4 de la página 17 vemos un ejemplo con 5 pistas creadas. Los colores nos ayudan a distinguir el tipo de pista, así las pistas *blancas* corresponden a *acompañamiento*, las *verdes* son las pistas de *Melodía*, y las *azules* las de *bajo*.

2.3.4. Creando bloques

Para GENARO, una división horizontal o en el tiempo es un *bloque*. Un conjunto de sonidos que tienen sentido musical por si mismos. El usuario puede decidir crear 1 o varios bloques para una pieza GENARO, que en su conjunto formarán la pieza que compondrá GENARO. Para crear un nuevo bloque en el proyecto, has de elegir primero el número de compases que quieres para ese bloque, mediante el selector que hay sobre el botón *Crear Bloque*, y después has de pinchar sobre el botón *Crear Bloque*. Un ejemplo con un bloque ya creado lo podemos ver en la figura 2.5 de la pgina 18. El color del bloque es rojo si este no ha sido inicializado, o el usuario ha decidido silenciarlo.

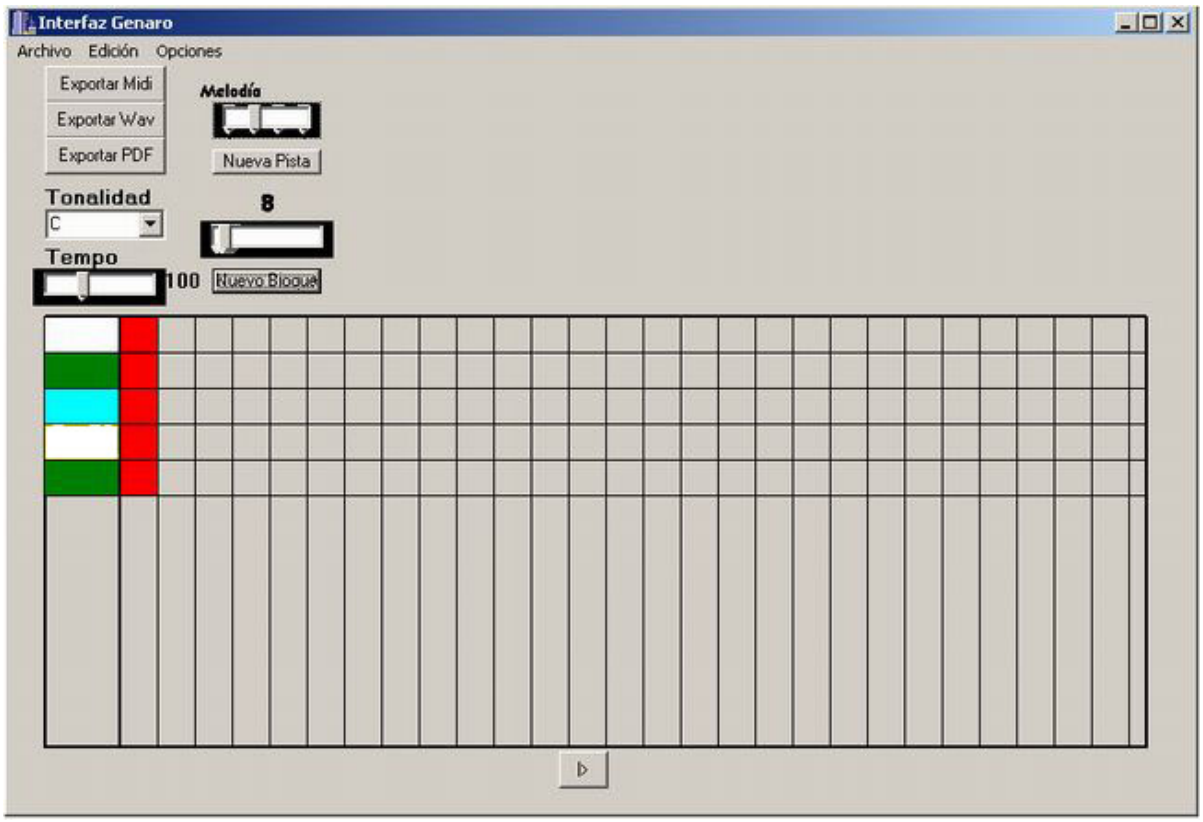


Figura 2.5: Ejemplo de proyecto con un bloque creado

2.3.5. Editando la cabecera de una pista

Como ya dijimos antes, una pista es un conjunto de sonidos asignados a un instrumento. Una pista esta formada por varios sub-bloques, que son la intersección entre una pista y un bloque. Podemos cambiar algunas opciones generales de una pista si pinchamos con el ratón sobre su cabecera, que corresponde al primer recuadro de la pista. Tras pinchar en este recuadro aparecerá ante nosotros un nuevo cuadro de opciones, situado arriba a la derecha, como se muestra en la figura 2.6 de la página 20. En este nuevo cuadro podemos distinguir varios elementos:

1. Una etiqueta que nos recuerda que pista estamos modificando (en este caso la *pista 0*).
2. Una etiqueta que nos recuerda que tipo de pista estamos modificando (en este caso una pista de *Acompañamiento*).
3. Una opción para silenciar toda la pista, de tal manera que la pieza creada prescindiría de reproducir esta pista.
4. Una lista de instrumentos midi, para poder elegir con cual interpretar la pista.
5. Un botón *Guardar* para guardar los cambios realizados en la pista.

.

2.3.6. Editando un sub-bloque de una pista

Para editar un sub-bloque de un pista, debemos pinchar sobre el sub-bloque correspondiente. Dependiendo del tipo de pista en la que estemos editando el sub-bloque podremos dar unos parámetros u otros, pero todos los sub-bloques tienen unas opciones en común, las *Opciones Generales*, que son las que aparecen en la figura 2.7 de la página 21.

En este recuadro de opciones generales podemos distinguir varias partes:

1. Una etiqueta que nos recuerda el número de compases del bloque al que pertenece.
2. Dos etiquetas que nos informan de a que pista y a que bloque pertenece el sub-bloque que estamos editando.
3. Una opción para silenciar el sub-bloque, y no ser interpretado en la pieza.
4. Un botón para guardar los cambios realizados en el sub-bloque
5. Un botón para generar el fragmento musical correspondiente a este sub-bloque, que es obligatorio generar antes de componer la pieza final. Dependiendo del tipo de pista en la que nos encontremos, tendremos que haber dado algunos parámetros u otros al programa.
6. Un botón para reproducir el fragmento musical perteneciente a este sub-bloque.

A continuación veremos las opciones de cada tipo de pista.

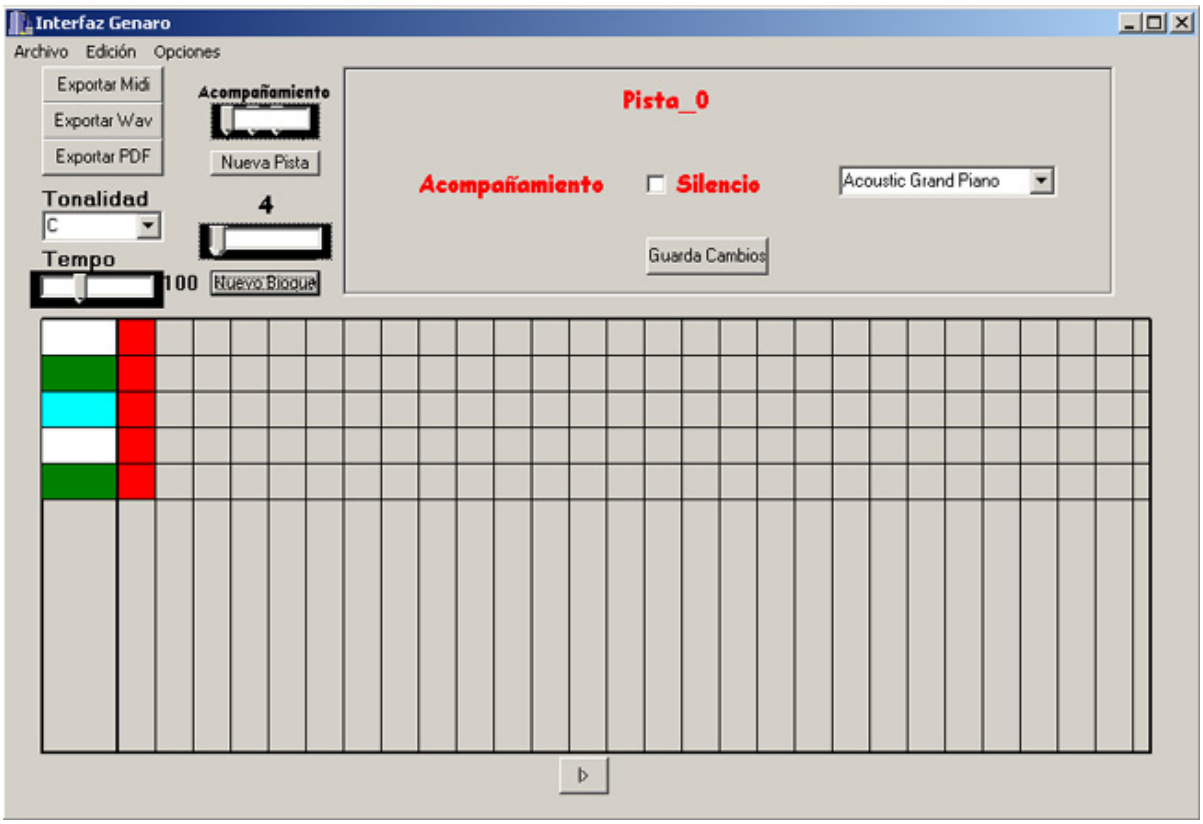


Figura 2.6: Editando las opciones de una pista

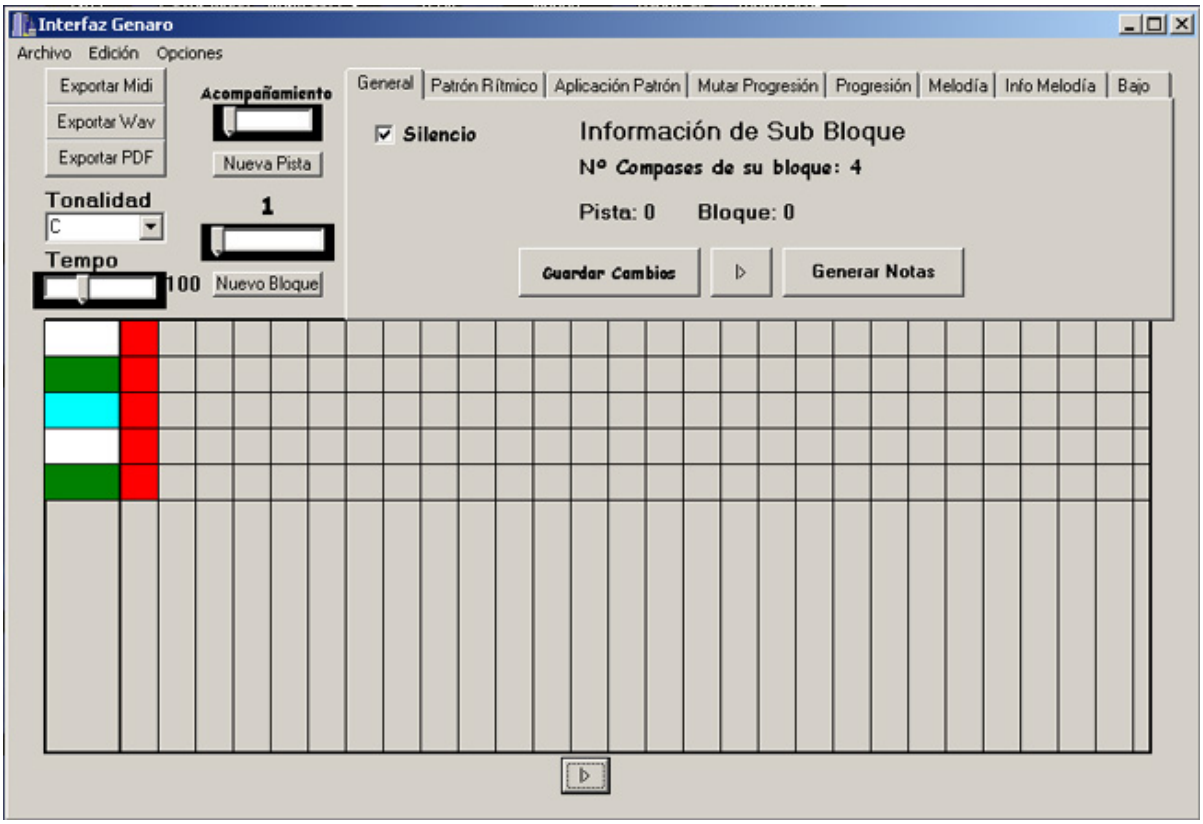


Figura 2.7: Editando las opciones generales de un sub-bloque

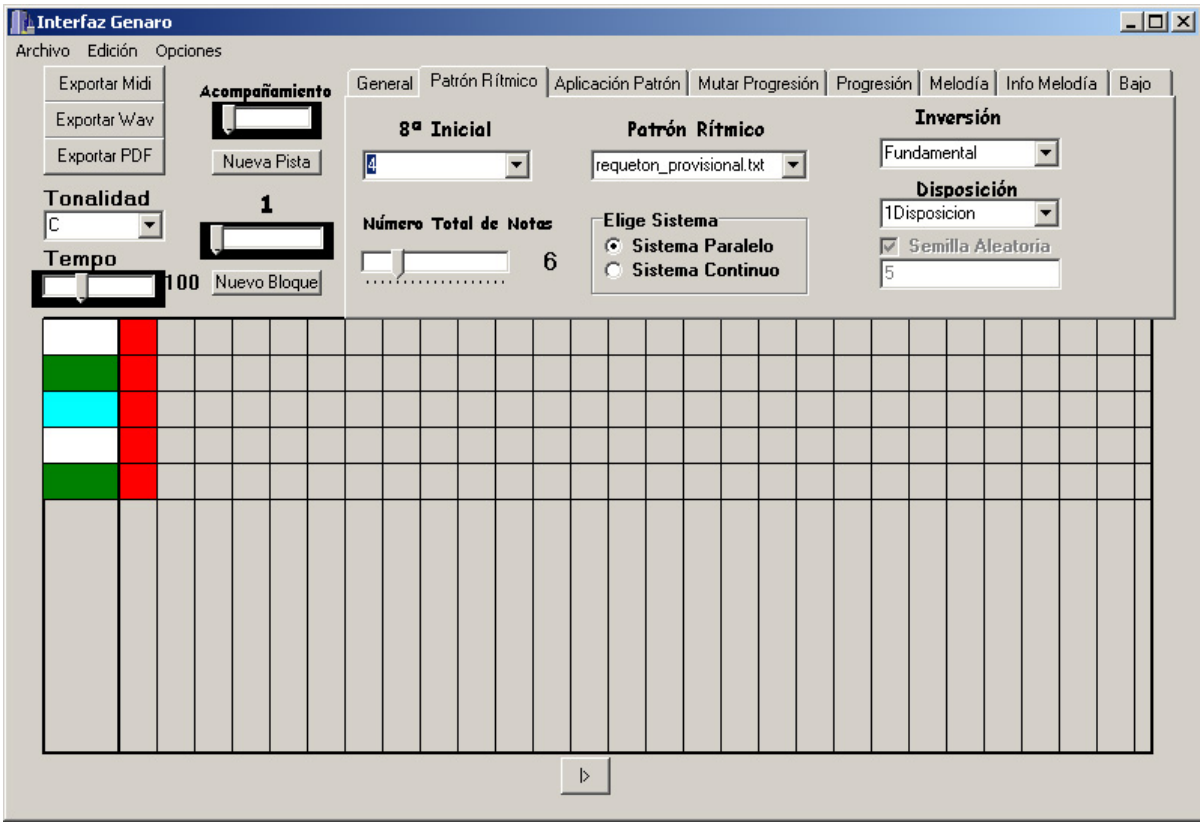


Figura 2.8: Editando las opciones de patrón rítmico de un sub-bloque

Editando un sub-bloque de acompañamiento

Un sub-bloque de acompañamiento ha de tener asignado una progresión para poder generar su parte correspondiendte. También es aconsejable haber modificado los parámetros que vienen por defecto. Los parámetros que se deben modificar están agrupados en 2 tipos, los parámetros correspondientes al *Patrón Rítmico* y a su aplicación, y a los correspondientes a la *progresión*.

1. Si vamos a modificar los parámetros correspondientes al *Patrón Rítmico*, tendremos ante nosotros una imagen como la mostrada en la figura 2.8 de la página 22. Los parámetros que se pueden modificar en este cuadro son:

Patrón Rítmico: es una lista desplegable que te permite elegir entre los patrones disponibles, algunos se distribuyen con GENARO, mientras que otros los puedes crear tú mismo usando el *Editor de Pianola* que viene con GENARO. Para más información sobre el editor de pianola, o de como modificar un patrón rítmico existente, consulta la sección dedicada al editor de pianola, sección 2.4 de la páginaedipianola.

8^a Inicial: consiste en una lista desplegable que nos permite elegir entre un nmero comprendido entre 1

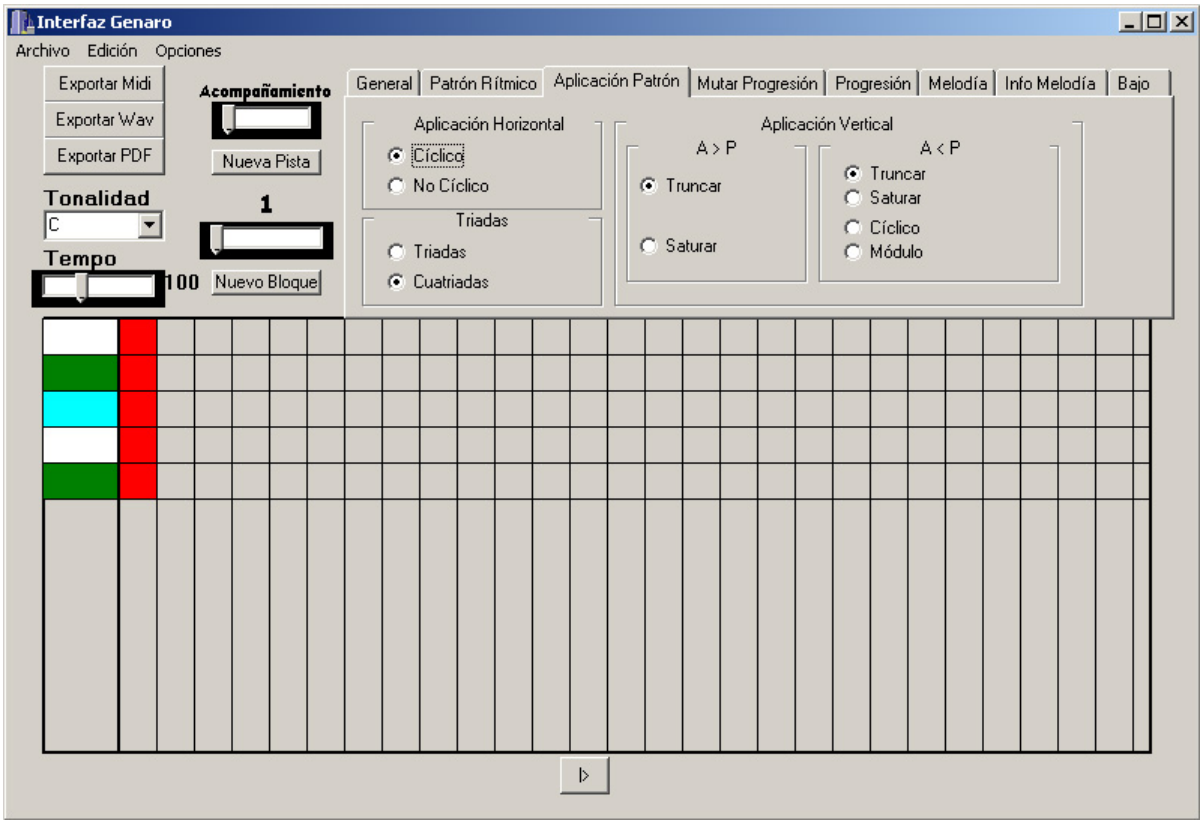


Figura 2.9: Editando las opciones de aplicación del patrón rítmico de un sub-bloque

y 7. La octava aconsejable es la 4, con una octava más baja obtendrás un sonido más grave, mientras que si escoges una octava mayor, el sonido será más agudo.

Número total de notas: es el número de notas que pueden sonar simultáneamente en la pieza, es aconsejable usar un valor próximo a 6.

Sistema: Existen 2 sistemas para generar el fragmento, cada uno de ellos tiene sus opciones independientes. Estos sistemas son el *sistema paralelo*, que te pide que elijas *Inversión* y *Disposición*, mientras que el *sistema continuo* te permite especificar una semilla en lugar de generarla aleatoriamente.

2. Para modificar los parámetros correspondientes a la aplicación del patrón, nos encontraremos ante un cuadro de opcioens como el mostrado en la figura 2.9 de la página 23. Aquí podemos modificar los siguientes parámetros:

Aplicación Horizontal: podemos elegir entre realizar la aplicación horizontal del patrón de manera *Cíclica* o *No cíclica*.

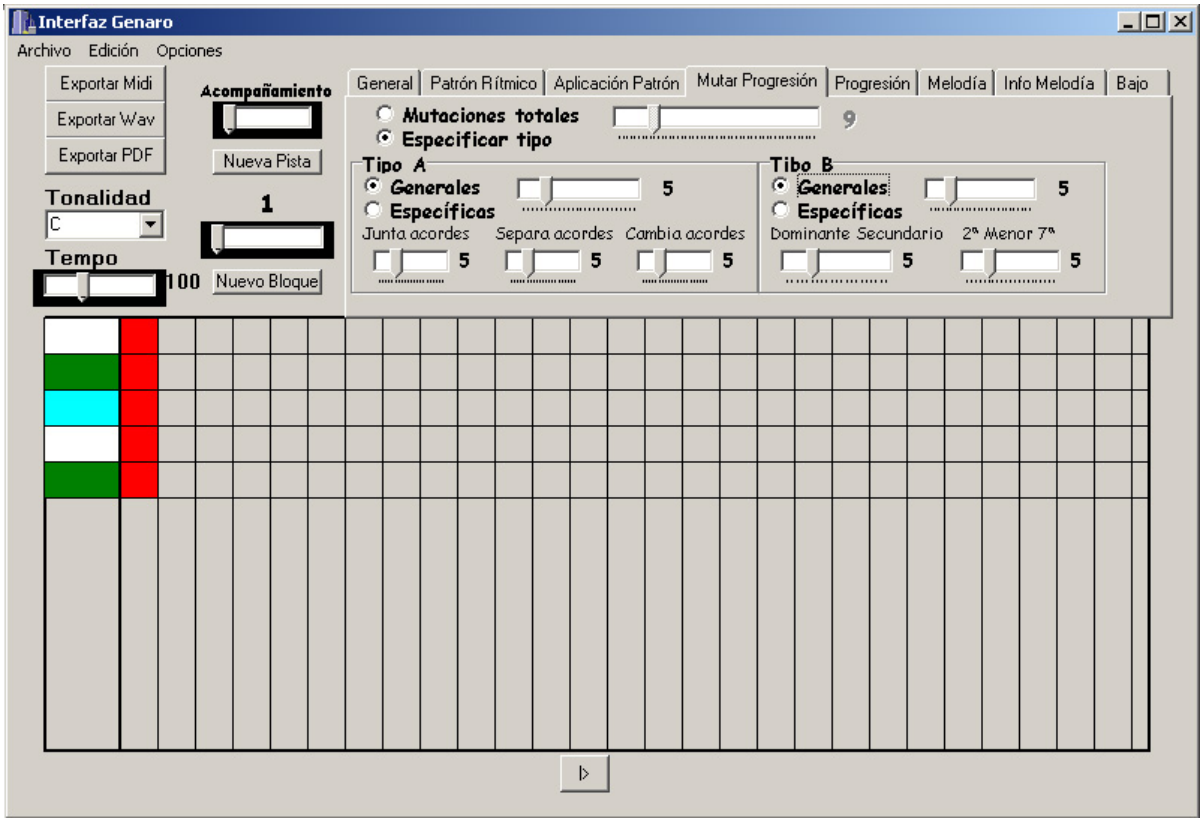


Figura 2.10: Editando las opciones de mutación de una progresión

Aplicación Vertical: aquí debemos especificar que acciones tomará para 2 casos, cuando hay mas acordes que patrones, en cuyo caso tendremos que especificar si queremos *truncar* o *saturar*, y el caso opuesto, donde tendremos que especificar si queremos *truncar*, *saturar*, *cíclico* o *módulo*.

Triadas: tendremos que especificar si preferimos *triadas* o *cuatriadas*.

3. Antes de crear una progresión tenemos que especificar el número de mutaciones que queremos, para ello vamos al cuadro *Mutar Progresión*. Existen 5 tipos distintos de mutaciones, *Juntar Acordes*, *Separar Acordes*, *Cambiar Acordes*, *Dominantes Secundario* y *Segunda Menor Séptima*. Estas mutaciones están a su vez agrupadas en 2 grandes tipos, así pues las 3 primeras pertenecen al *Tipo A*, y las 2 últimas al *Tipo B*. El usuario puede elegir entre realizar un número de mutaciones independientemente del tipo, (opción *Mutaciones Totales*), o bien especificar cuantas mutaciones de cada tipo prefiere, mediante las barras de desplazamiento que se muestran en la figura 2.10 de la página 24.

4. Para crear una progresión, o usar una ya creada, tenemos que irnos al cuadro *Progresión*, tras esto tendremos una imagen similar a la que aparece en la figura 2.11 de la página 25. En este cuadro encontraremos las siguientes opciones:

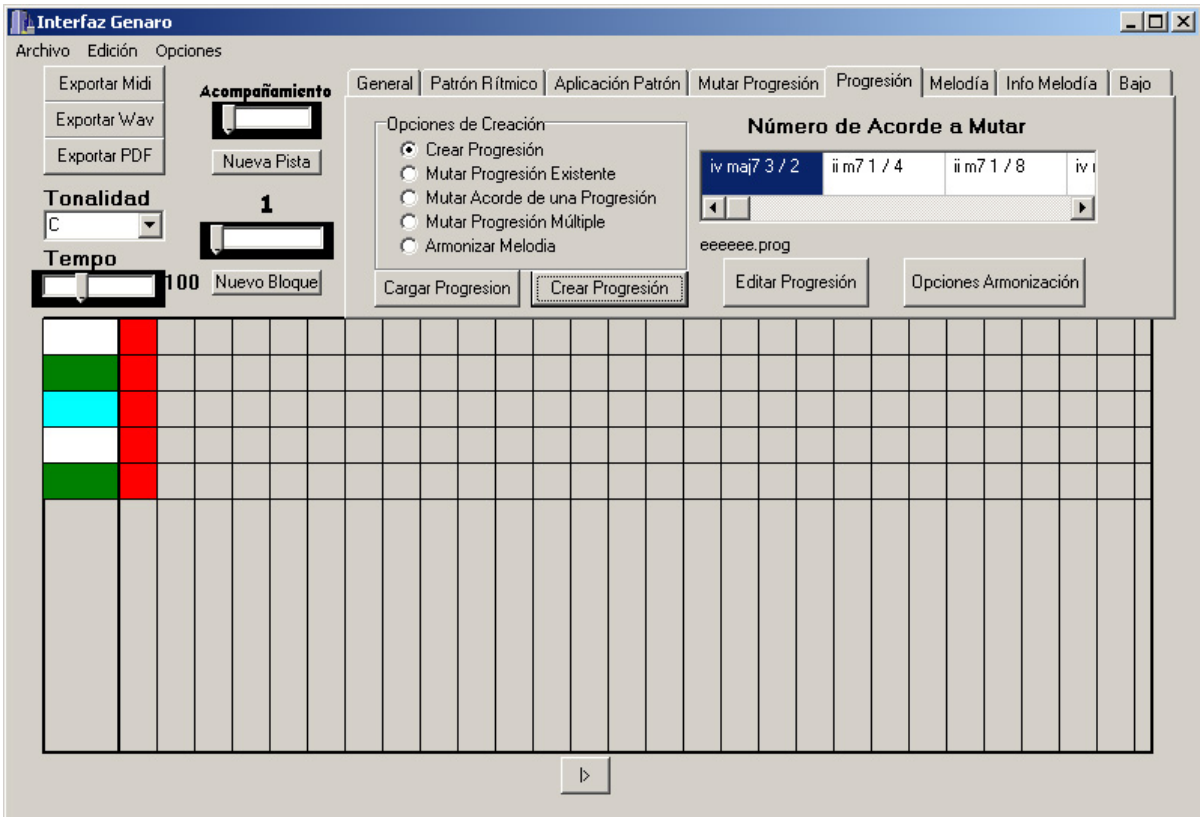


Figura 2.11: Editando las opciones de creación de una progresión

- *Opciones de Creación:* nos permite elegir entre las distintas formas de crear una progresión. las analizaremos mas adelante en profundidad.
- *Cargar Progresión:* si preferimos usar una progresión creada anteriormente, tendremos que pulsar este botón, y especificar cual es la progresión que queremos usar.
- *Crear Progresión:* tras escoger la opción de creación, si pulsamos este botón crearemos una nueva progresión con el nombre que especifiquemos.
- *Progresión:* En un grid aparecerá la progresión con la que estemos trabajando actualmente, con su nombre debajo. Hay que tener en cuenta que el acorde que tenemos seleccionado será el acorde en el que realizaremos la mutación en caso de escoger la opción *Mutar acorde de una progresión*.
- *Editar Progresión:* si pulsamos este botón abriremos una nueva ventana para editar la progresión con la que estamos trabajando. La ventana que aparece se muestra en la figura 2.12 de la página 27. Nos permite crear nuevos acordes y sustituir los existentes, o añadirlos delante o detras de los mismos.
- *Opciones de Armonización:* Al igual que en el caso anterior, este botón creará una nueva ventana que nos permitirá elegir entre distintas opciones de armonización, tal y como se muestra en la figura 2.13 de la página 27.

Con respecto a las opciones de creación de progresión, podemos distinguir las siguientes:

- *Crear Progresión:* crea una nueva progresión con los parámetros de mutación anteriormente especificados
- *Mutar Progresión Existente:* genera una nueva progresión a partir de una progresión ya dada, aplicando las mutaciones especificadas anteriormente.
- *Mutar Acorde de una Progresión:* muta el acorde seleccionado en la progresión aplicando las mutaciones especificadas.
- *Mutar Progresión Múltiple:* permite crear una progresión tomando la semilla de una progresión ya existente
- *Armonizar Melodía:* Permite coger una pista de melodía para crear una progresión con ella.

No debemos olvidar que una vez hayamos terminado de especificar todas las opciones necesarias, tenemos que ir a las opciones generales y generar las notas correspondientes al sub-bloque actual (para ello pulsa el botón *Generar Notas*). Un sub-bloque que no está silenciado se dibuja de color *azul*, en vez del color *rojo* con el que aparece inicialmente.

Editando un sub-bloque de melodía

Una melodía precisa de una pista de acompañamiento para poder componer su fragmento. Es necesario que al menos hayas creado la progresión de la pista de acompañamiento que va a tomar como referencia la melodía para poder componer su fragmento. Los distintos cuadros que nos permiten alterar los parámetros de creación de una melodía son los siguientes:

1. los parámetros que se hallan en el cuadro *Melodía* (mostrado en la figura 2.14 de la página 28) van a determinar la melodía que se va a generar. Los parámetros que aquí se encuentran son:

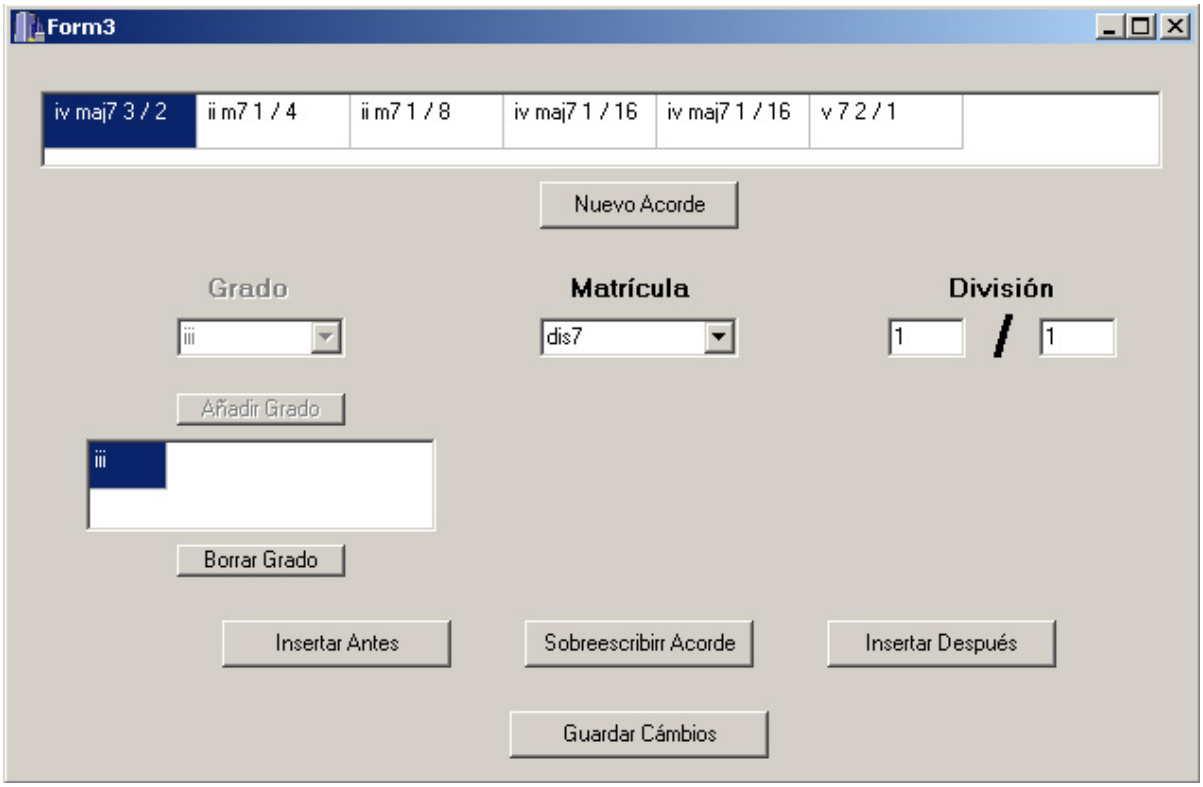


Figura 2.12: Editando una progresión

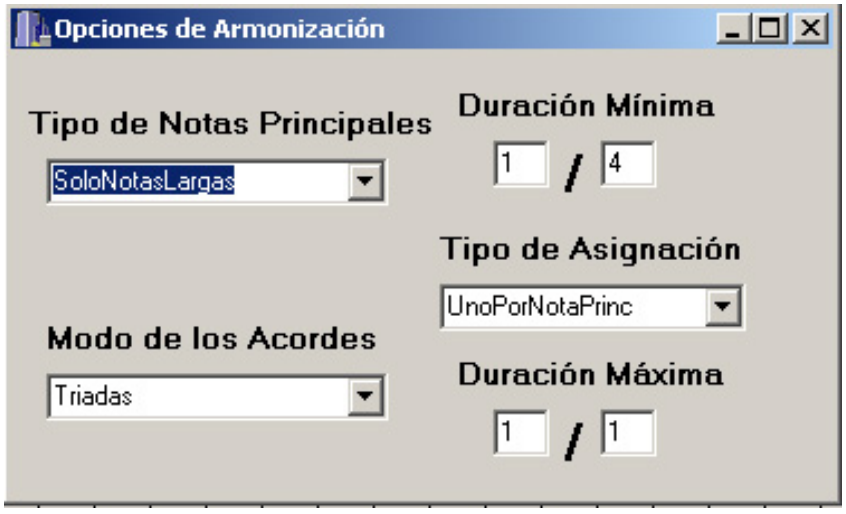


Figura 2.13: Opciones de armonizacion

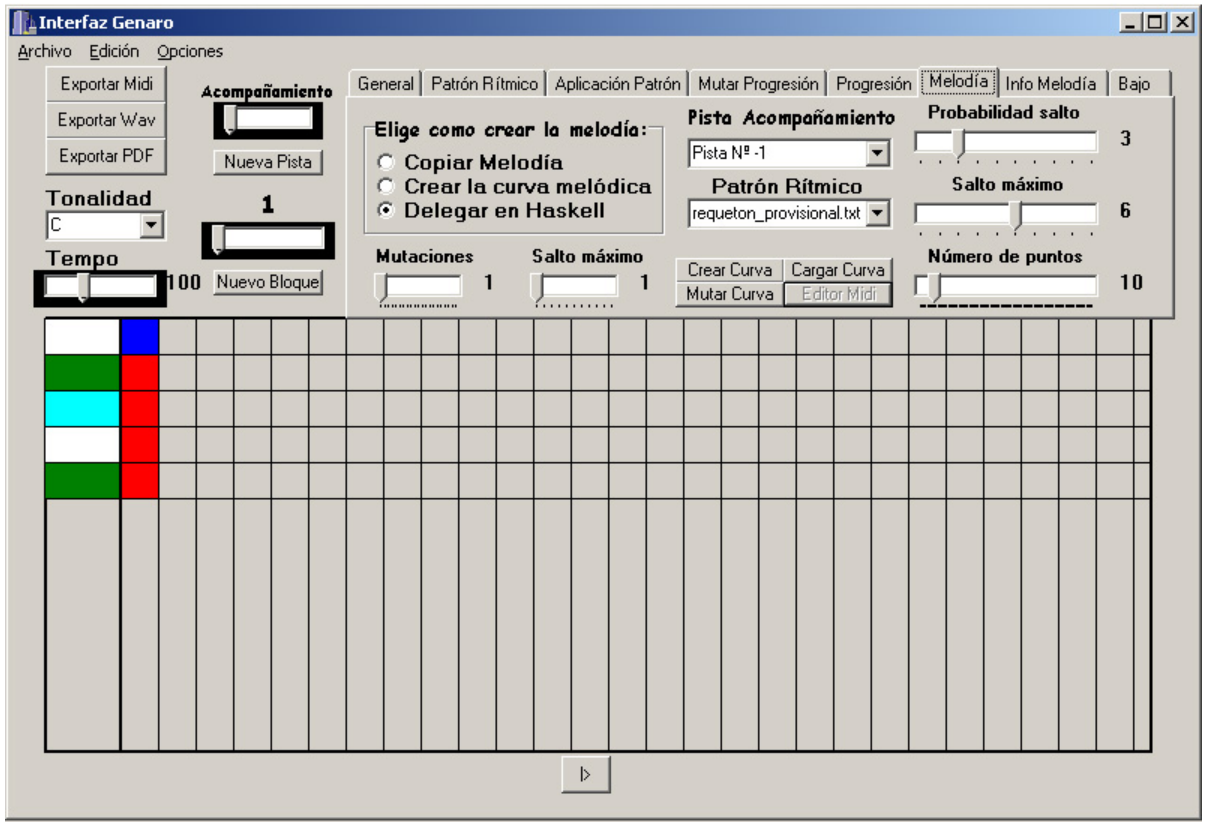


Figura 2.14: Editando una melodía

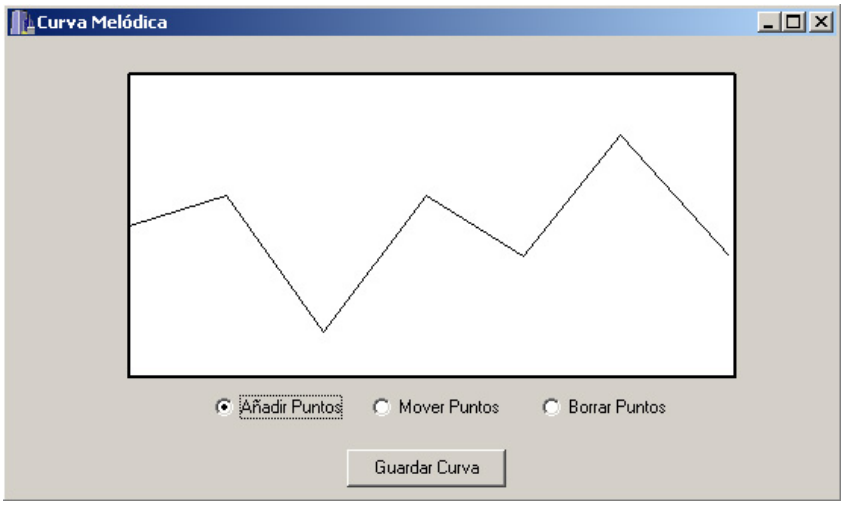


Figura 2.15: Editando una curva melódica

- *Elige como crear la melodía:* que tiene tres opciones, *Copiar Melodía* de otro sub-bloque de melodía, creando una curva melódica o delegando en haskell para que cree una curva melódica.
- *Pista de Acompañamiento:* en esta lista tenemos todas las pistas de acompañamiento existentes, tenemos que seleccionar una.
- *atrón Rítmico:* una lista con los patrones existentes, para tomar como base para crear la melodía.
- *Botón Crea Curva:* que dependiendo de la opción marcada en *Elige como crear la melodía* usará los parámetros establecidos a su derecha para llamar a haskell y crear una nueva curva, o bien arrancará una nueva ventana como la mostrada en la figura 2.15 de la página 29 para que el usuario pueda crear o editar la suya propia.
- *Botón Cargar Curva:* si tenemos intención de usar alguna curva ya creada anteriormente. Hay que tener en mente que las curvas se crean con nombres automáticos, con 2 números que corresponden a la pista y al bloque.
- *Botón Mutar Curva:* para mutar la curva existente, usando los parámetros que hay a la izquierda de este botón.

2. Otras opciones que podemos especificar para crear la melodía están en el cuadro *Info Melodía*, como se muestra en la figura 2.16 de la página 30, pueden ser también alteradas, para producir un resultado distinto, estas opciones son 4 barras de desplazamiento que nos permiten elegir entre *velocidad de trino*, *Dividir notas*, *Alargar notas* y *Trinos*.

Editando un sub-bloque de Bajo

El bloque de *Bajo* tiene sólo un cuadro para cambiar parámetros, es el mostrado en la figura 2.17 de la página 31. Los más relevantes son:

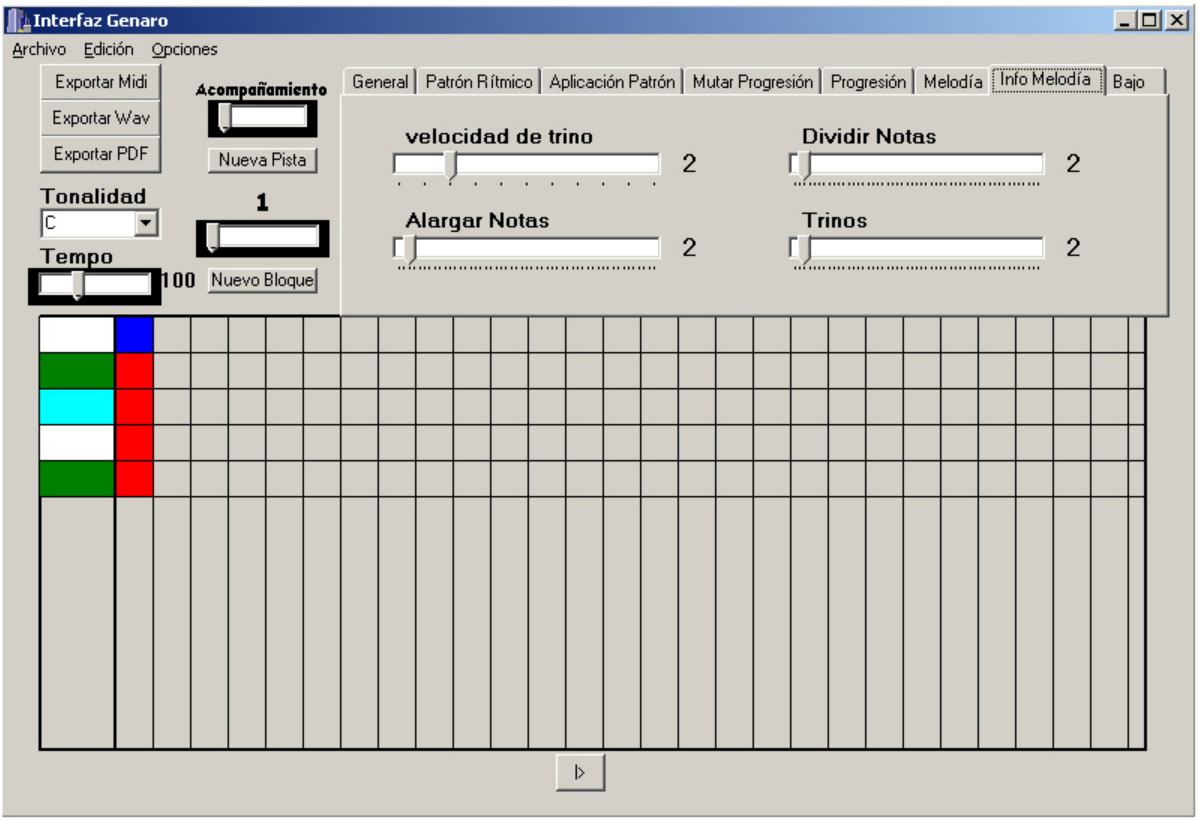


Figura 2.16: Editando una melodía, parte 2

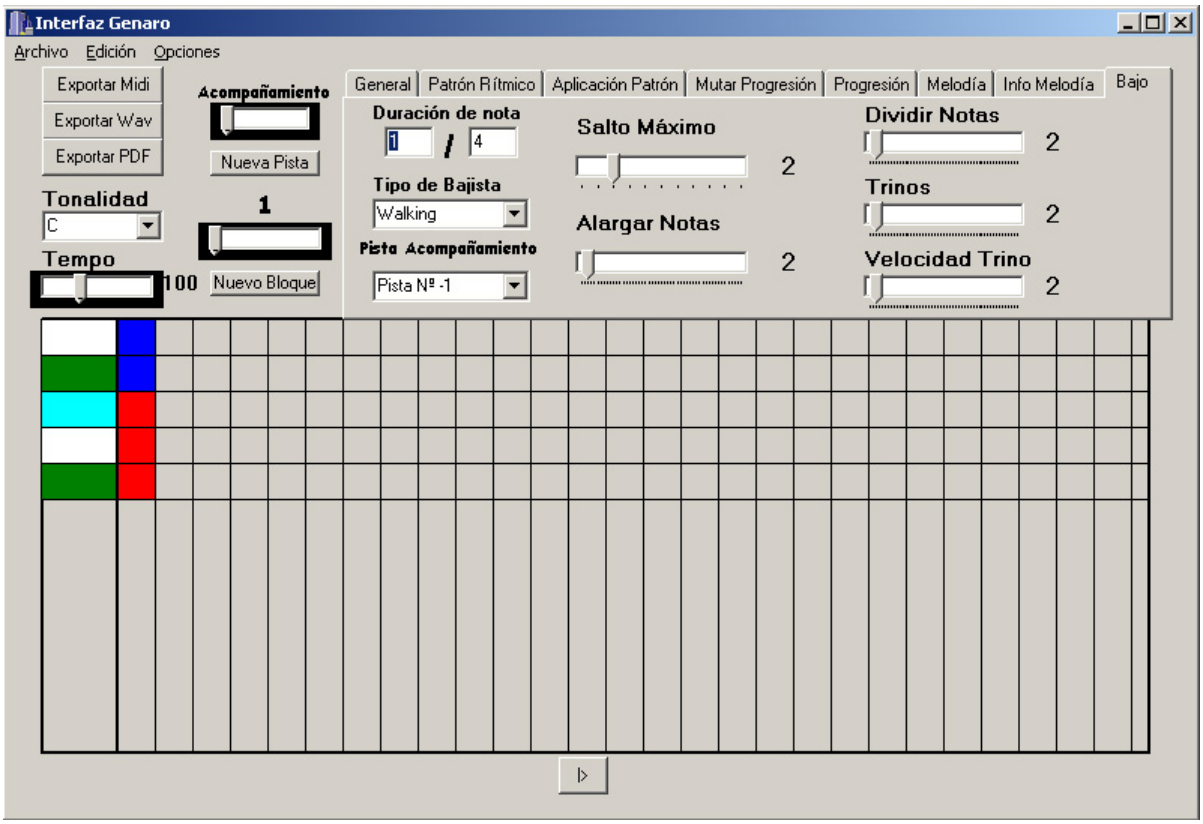


Figura 2.17: Editando una sub-bloque de Bajo

- *Pista de Acompañamiento:* en esta lista tenemos todas las pistas de acompañamiento existentes, tenemos que seleccionar una.
- *Tipo de bajista:* una lista de 3 bajistas, cada uno compone de una manera muy distinta al otro.
- *Duración de la nota:* que es una fracción de números naturales
- *Otros Parámetros:* sin ser tan importante como estos, existen otros 5 parámetros que nos permiten configurara a nuestro gusto el bajo, son *Salto Máximo*, *velocidad de trino*, *Dividir notas*, *Alargar notas* y *Trinos*.

2.3.7. Creando la pieza final

Si ya has terminado de inicializar todos los sub-bloques correspondientes a tu proyecto, tan sólo queda crear el midi final. Antes de esto tienes que especificar la *tonalidad* con la que quieres crear tu pieza, (viene en notación americana), está en la lista que hay a la izquierda del todo, y justo debajo hay un selector para ponerle el *Tempo* que quieras a tu obra. Si estás de acuerdo con la tonalidad y tempo que has elegido, presiona el botón *Exportar Midi*, y tras unos segundos tu midi estará creado. Se halla en el directorio raíz del programa, con el nombre de *musica-genara.mid*, pero te aconsejamos que lo reproduzcas diréctamente desde el interfaz, ya que está enlazado con *Timidity*, un excelente programa de reproducción de midis, con el que escucharás tu pieza con gran calidad. Existen también opciones de reproducción en genaro, pero las vamos a especificamos en la sección 2.3.8 de la página 32.

2.3.8. Opciones de reproducción

El usuario puede editar las opciones de reproducción de midi del programa *Timidity* pulsando sobre el menú *Opciones* y posteriormente pulsando en *Reproducción*. Tras esto aparecerá una nueva ventana como la mostrada en la figura 2.18 de la página 33. Esta nueva ventana te permite modificar los siguientes parámetros de reproducción:

- *Amplificación de volumen:* permite modificar el volumen con el que *Timidity* va a reproducir/exportar la pieza.
- *Chorus Effects:* permite activar/desactivar los coros, eligiendo el modo. También se puede especificar el level mediante la barra de desplazamiento que hay bajo esta opción.
- *Reproducir con otro instrumento:* es una lista que permite elegir un *patch* para reproducir todas las pistas con este instrumento.
- *Frecuencia de muestreo:* te permite cambiar el número de muestras por segundos
- *Reverb Effects:* puedes activar y desactivar el efecto reverb, y ajustar el nivel del mismo.
- *Usar Delay Effects:* para activar el efecto delay, con opciones de usarlo a la izquierda, derecha o rotarlo.
- *Botón Valores por defecto:* al pulsar este botón pone todos los parámetros al valor por defecto.
- *Botón Aplicar Cambios:* guarda los cambias realizados en las opciones de reproducción. Si no pulsas este botón se descartarán los cambios hechos.

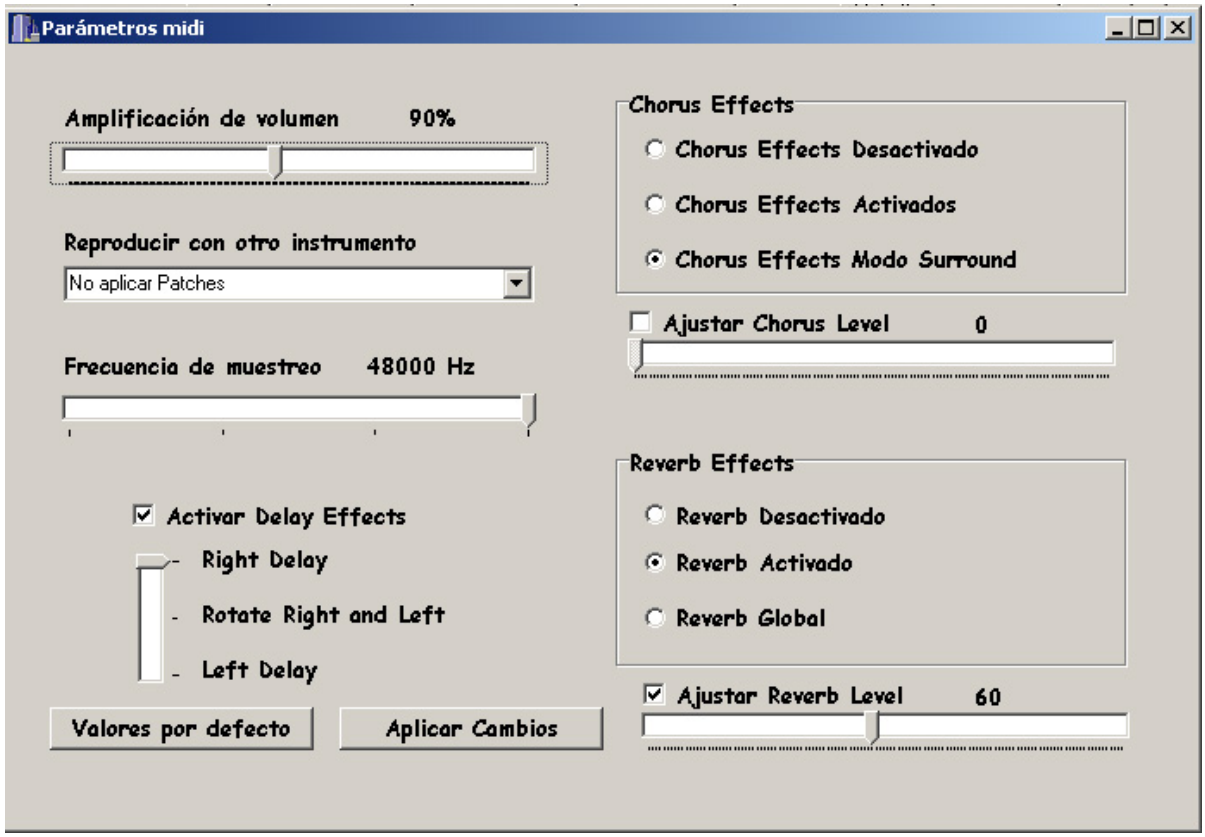


Figura 2.18: Opciones de reproducción para el midi

2.3.9. Exportando a wav

Exportar a wav es una opción que permite una vez generado el archivo midi, crear un archivo wav. Para ello llama a *timidity* con los parámetros de reproducción especificados por el usuario en *Opciones de Reproducción*, y crea un fichero wav con el nombre de *musica-genara.wav*, que se encuentra en el directorio raíz del programa. Para exportar el fichero midi a wav, tan sólo pulsa sobre el botón *Exportar a Wav* que se encuentra debajo del botón *Exportar a Midi*.

2.3.10. Guardando y cargando proyectos

GENARO permite guardar un proyecto para poder trabajar con el posteriormente. Para ello dirígete al menú *Archivo* y escoge la opción *Guardar*. El programa te preguntará por el nombre con el que quieres guardar el proyecto, y posteriormente lo guardará con el nombre proporcionado.
Para cargar un archivo, dirígete al menú *Archivo*, y escoge la opción *Cargar*. Escoge el archivo de proyecto que desees, y continúa trabajando donde lo dejastes.

2.4. El editor de pianola

El editor de pianola es una utilidad que acompaña a GENARO para la edición y creación de patrones rítmicos. Un patrón rítmico es una estructura capaz de organizar las voces de un acorde en el tiempo. Es un componente no aleatorio.
Para ejecutar el editor de pianola, has de pulsar sobre el menú *Edición* del programa principal, escoger la opción *Editor de pianola*, y te aparecerá una nueva ventana como la mostrada en la figura 2.19 de la página 35.

2.4.1. Creando un nuevo proyecto

Al igual que con el programa principal, para trabajar con el editor de pianola tenemos que crear un nuevo proyecto, o bien, cargar uno existente. Para crear un nuevo proyecto, sencillamente dirígete al menú *Archivo*, y escoge la opción *Nuevo*. Tras esto ya puedes empezar a trabajar en el patrón rítmico.

2.4.2. Creando voces en el editor de pianola

Para crear una nueva voz en el editor de pianol, tan sólo has de pulsar sobre el botón *Nueva Voz* que se encuentra arriba a la izquierda. Cada vez que pulses este botón, una nueva fila aparecerá en el editor. Si creas más filas de las que es capaz de mostrar el editor, podrás desplazarte por ellas mediante el uso de la barra de desplazamiento vertical que hay a la izquierda del todo.

2.4.3. Aumentando la resolución del editor

Como habrás observado, el editor de pianola posee una barra de desplazamiento con las palabras *Zoom In* y *Zoom out*. Al desplazar la barra en un sentido o en otro, aumentamos o disminuimos el tamaño con el que vemos la rejilla.

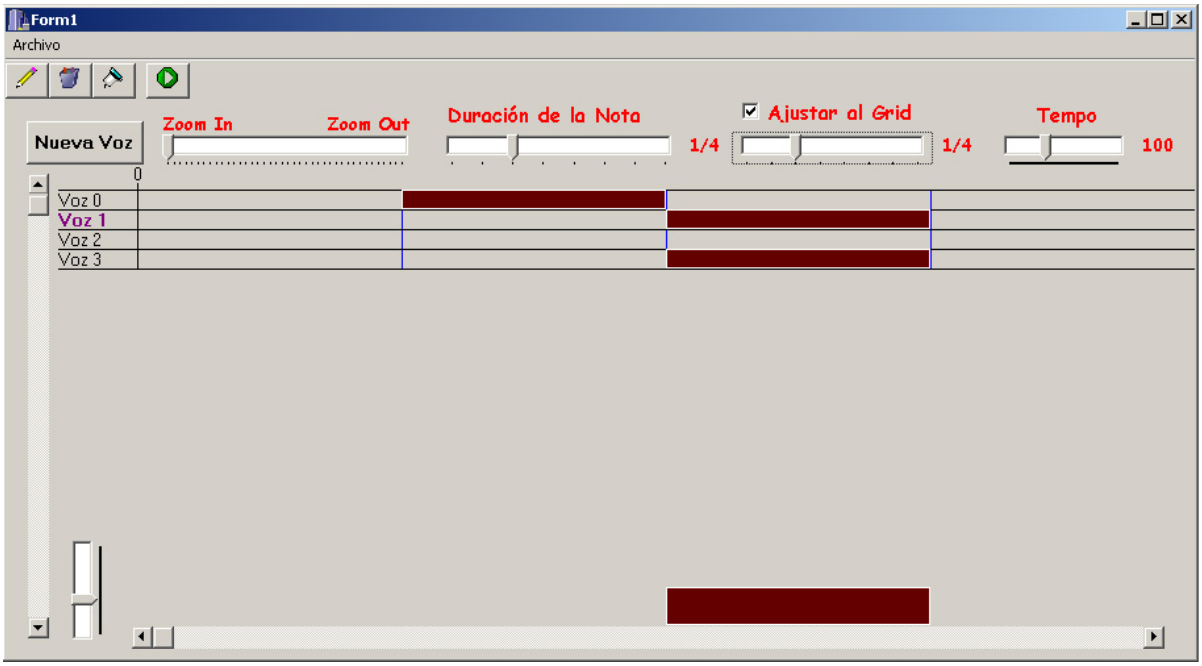


Figura 2.19: El editor de pianola

2.4.4. Eligiendo la duración de la nota que vamos a añadir

Cuando estemos en *modo edición*, cada vez que pulsemos sobre la rejilla intentaremos añadir una nueva nota. Esta nota tendrá la duración determinada por la barra de desplazamiento *Duración de la nota*.

2.4.5. El grid, divisiones para facilitar la edición

El editor de pianola permite crear separaciones de la duración especificada por el usuario, para hacer mas cómoda la edición. Para ello debemos desplazar la barra *Ajustar al Grid*. Posee también una opción para ajustar la nota al grid, marcando la casilla que hay en *Ajustar al Grid*. Con esta opción marcada, cualquier nota que se añada al patrón, se pondrá en la zona a la que pertenezca la división especificada en *Ajustar al Grid*.

2.4.6. El menú edición

Justo debajo del menú del editor de pianola, encontramos 4 botones. Su función es la siguiente:

Lápiz: Es el botón edición, mientras este botón esté pulsado, cada vez que pinchemos sobre la rejilla intentaremos insertar una nueva nota

Papelera: Es el botón eliminar. Nos permite borrar notas de la rejilla pinchando sobre ellas.

Escaner: Es el botón examinar. Nos permite seleccionar una voz sin realizar ningún cambio en el sistema. También nos permite modificar el *Velocity* de cada nota.

Play: Es el botón reproducir, nos permite escuchar como sonaría el patrón que estamos componiendo. Para cambiar el tempo de esta reproducción debemos mover la barra de desplazamiento llamada *Tempo*

2.4.7. El velocity

El velocity corresponde por ejemplo a la *fuerza* con la que se pulsa una tecla en un piano, podemos modificar el velocity de cada nota escogiendo el *modo examinar*, y pinchando sobre la parte inferior de la rejilla, subiendo o bajando el rectángulo que aparece, observese que dependiendo de su altura el color de este rectángulo y de la nota a la que pertenece cambian.
Podemos ajustar un velocity por defecto moviendo el selector que hay debajo de la rejilla.

2.4.8. Guardando y cargando

Si queremos guardar el patrón, debemos pulsar sobre el menú *Archivo*, y en la opción *Guardar*. El programa nos pedirá con que nombre queremos guardar este patrón. Es importanto que los patrones se guarden en el subdirectorio del programa *PatronesRitmicos*, ya que ahí es donde los busca GENARO.
Para cargar un patrón debemos dirigirnos al menú *Archivo*, y escoger la opción *Cargar*.

Capítulo 3

Generador de progresiones de acordes

3.1. Introducción

El módulo generador de acordes se encarga de generar y modificar *progresiones de acordes*. Para comprender como funciona y para que sirve primero debemos entender el concepto de progresión de acordes.

3.2. Acordes y progresiones de acordes

Recordemos que a un nivel más alto de abstracción un acorde se puede entender como una agrupación de alturas musicales que se considera que tienen un significado conjunto. Ese será el grado de abstracción al que trabajaremos en esta fase de la generación de música, y llamaremos *cifrado* a la representación de los acordes a este nivel de abstracción.

Como este concepto de acorde solamente se refiere a la altura de los sonidos, emparejaremos los cifrados con duraciones (figuras), que expresaran el espacio de tiempo que ocupará la serie de notas, es decir, la música, en la que se concretará el acorde. Y a las listas de cifrados emparejados con figuras las llamaremos *progresiones de acordes*. Estas listas expresan la sucesión de los acordes a lo largo del tiempo. En una fase posterior de la generación de música se realizará la citada correspondencia entre parejas (cifrado, figura) y músicas, por tanto entre progresiones de acordes y músicas (la música que corresponde a una progresión es la que resulta de suceder en el tiempo las músicas que corresponden a cada par perteneciente a la progresión, desde el primero al último). Las diversas maneras en que se puede hacer esta correspondencia permitiran generar músicas muy diversas a partir de una misma progresión de acordes.

3.3. Impacto de los acordes en el resto de la música

Según las reglas de la Armonía, los acordes establecen una jerarquía de sonidos, definiendo un *contexto armónico* que a su vez implica una función que valora alturas musicales haciéndole corresponder

un entero que llamamos estabilidad de esa altura en ese acorde. Esta función se empleará para elegir las notas del bajo y la melodía que se compongan a partir de un acompañamiento, teniendo mayor probabilidad de aparecer en una melodía o en un bajo las notas más estables.

El contexto armónico de un acorde esta determiando no sólo por el acorde sino también por su relación con un eje o contexto global que llamamos *tonalidad*, y que define la estabilidad del acorde dentro de este contexto global, y cuáles son los acordes que es más probable que le sigan en una progresión. Así que las progresiones de acordes siguen una cierta lógica, estando determinado en cierto medida un acorde por los acordes anteriores a él y por los acordes que le siguen en la progresión. Esto es debido a que, según el sistema de armonía empleado en GENARO, la música es un juego entre estabilidad e inestabilidad, los acordes inestables generan un movimiento o *tensión* que termina por liberarse o *resolverse* sobre acordes más estables. La forma en que GENARO simula esto se explica en la sección siguiente.

3.4. Algoritmo de generación de progresiones de acordes

El algoritmo consiste en partir de una progresión inicial, que llamamos *progresión semilla*, ajustarla para que tenga el número de compases especificado como entrada del algoritmo, y aplicarle varias transformaciones aleatorias (*mutaciones*) hasta conseguir la progresion resultado. Los pasos a seguir son los siguientes:

1. Elegir una progresión semilla:
   - Se elige de entre una base de datos de progresiones típicas, por lo que esta semilla no se genera aleatoriamente, aunque si que se elige aleatoriamente.
   - Las progresiones semillas que hay actualmente en la base de datos duran entre 1 y 4 compases.
   - La aplicación de una cantidad suficiente de mutaciones garantiza que las progresiones genaradas sean variadas.
2. Ajustar el número de compases:
   - Para ajustar el número de compases de la progresión semilla se restringe las progresiones candidatas a ser semilla a aquellas cuyo duración en número de compases sea divisor del número de compases que es entrada del algoritmo. Luego se multiplica el número de compases de todos los acordes de la semilla por la division del número de compases de entrada entre el número de compases de la semilla, para lograr una progresión semilla con el número de compases deseado.
   - Este sistema funciona porque en las progresiones semilla todos los acordes tienen la misma duración. Si tuvieran duraciones distintas se podría modificar fácilmente el algoritmo para lograr un encaje de duraciones de otra manera.
   - El hecho de contar con una progresión semilla que dura un compás garantiza que siempre se pueda lograr una progresión semilla ajustada a cualquier número de compases de entrada, ya que el 1 divide a cualquier número.
3. Mutar la progresión: Las mutaciones son transformaciones que respetan dos propiedades que cumplen todas las propiedades semilla. De esta manera mantenemos como invariante que las progresiones con las que trabajamos cumplen estas propiedades, y podemos afirmar que la progresión resultado también las cumple. Las propiedades son:

- Relación con la tonalidad: Todos los acordes de la progresión, o bien pertenecen a la tonalidad (en término técnicos, son diatónicos), o bien están estrechamente relacionados con ésta (el caso de los dominanetes secundarios y IIm7).
- Respeto del rítmo armónico: Esta es otra cuestión técnica que se puede resumir en que los acordes que se consideran más estables valorándolos en el contexto que establece la tonalidad, se sitúan en tiempos del compás que se consideran más fuertes.

Sin entrar en tecnicismos, se trata de modificar las progresiones sin que estas pierdan su musicalidad, ni su sentido al interpretarlas según las reglas de la armonía. Además las mutaciones no cambian el número de compases de la progresión, por lo que basta con ajustar el número de compases en la semilla para conseguir el número de compases deseado.

Las mutaciones son sustituciones de un acorde de la progresión por una lista de acordes, y son aleatorias en dos sentidos:

(a) La mutación a aplicar se elige aleatoriamente

(b) El acorde a mutar se elige aleatoriamente

Podemos expresar las mutaciones mediante reglas de reescritura, que suponemos indeterministas ya que éste módulo aplica las mutaciones aleatoriamente. Las mutaciones disponibles son *cambia_acordes*, *aniade_acordes*, *quita_acordes*, *dominante_secundario* y *iim7*:

```
cambia_acordes([]) = []
cambia_acordes([Ac| Acs]) = [acorde_equivalente(Ac)| Acs]
cambia_acordes([Ac| Acs]) = [Ac| cambia_acordes(Acs)]

aniade_acordes([]) = []
aniade_acordes([Ac| Acs]) = pareja_dividida(Ac) ++ Acs
aniade_acordes([Ac| Acs]) = [Ac| aniade_acordes(Acs)]

pareja_dividida(Ac) =
    [pon_duracion(Ac, D/2)
    ,pon_duracion(acorde_equivalente(Ac), D/2)] <== dame_duracion(Ac) == D


quita_acordes([]) = []
quita_acordes([Ac1, Ac2| Acs]) = pareja_unida(Ac1, Ac2) ++ Acs
        <== equivalentes(Ac1, Ac2) == true
quita_acordes([Ac| Acs]) = [Ac| quita_acordes(Acs)]

pareja_unida(Ac1, Ac2) =  [pon_duracion(acorde_equivalente(Ac1),D1+D2)]
        <== dame_duracion(Ac1) = D1
           ,dame_duracion(Ac2) = D2

dominante_secundario([]) = []
dominante_secundario([Ac1, Ac2| Acs]) = monta_dom_sec(Ac1) ++ [Ac2|Acs]
```

```
        <== candidato_dominante(Ac1, Ac2) == true
dominante_secundario([Ac| Acs]) = [Ac| dominante_secundario(Acs)]

monta_dom_sec(Ac) = [pon_duracion(Ac, D/4)
                    ,pon_duracion(acorde_dominante_de(Ac), D/4)
                    ,pon_duracion(Ac, D/2)
                     ]
        <== dame_duracion(Ac) = D

iim7([]) = []
iim7([Ac1, Ac2| Acs]) = monta_iim7(Ac1) ++ [Ac2|Acs]
        <== candidato_iim7(Ac1, Ac2) == true
iim7([Ac| Acs]) = [Ac| iim7(Acs)]

monta_iim7(Ac) = [pon_duracion(acorde_iim7_de(Ac), D/2)
                 ,pon_duracion(Ac, D/2)
                 ]
        <== dame_duracion(Ac) = D
```

3.5. Implementación

El lenguaje de implementación empleado en esta parte de GENARO fué SWI-Prolog. Al principio usábamos Sicstus Prolog pero terminamos pasándonos a SWI-Prolog principalmente porque es libre, y porque el algoritmo de generación de progresiones no necesita una gran potencia de proceso para ser implementado, y para implementar este algoritmo es para lo único que usamos Prolog. Además la vuelta atrás de Prolog se aprovecha para hacer indeterminista el algoritmo.
Como Prolog no tiene tipos se diseñaron una serie de predicados que definen las estructuras con las que trabajamos en este módulo, como por ejemplo las progresiones. Estos predicados tienen éxito cuando sus argumentos de entrada son términos con la estructura deseada, y no son llamados nunca, se desarrollaron solamente como especificación.
La representación de progresiones en Prolog es la siguiente:

```
es_progresion(progresion(P)) :- es_listaDeCifrados(P).
es_listaDeCifrados([]).
es_listaDeCifrados([(C, F)|Cs]) :- es_cifrado(C),es_figura(F)
                                  ,es_listaDeCifrados(Cs).

es_cifrado(cifrado(G,M)) :- es_grado(G), es_matricula(M).

es_matricula(matricula(M)) :-
        member(M, [mayor,m,au,dis,6,m6,m7b5, maj7,7,m7,mMaj7,au7,dis7]).

es_grado(grado(G)) :- es_interSimple(interSimple(G)).
es_grado(grado(v7 / G)) :- es_grado(grado(G)).
es_grado(grado(iim7 / G)) :- es_grado(grado(G)).
```

```
es_interSimple(interSimple(G)) :-
member(G, [i, bii, ii, biii, iii, iv, bv, v, auv, vi, bvii, vii]).
es_interSimple(interSimple(G)) :-
  member(G, [bbii, bbiii, auii, biv, auiii, auiv, bbvi, bvi, auvi, bviii, auviii]).
```

En éste módulo se trabaja manejando términos con esta estructura. El resultado último de la generación de progresiones un archivo de texto de nombre especificado en la entrada de Prolog, que contiene un término que cumple el predicado es_progresion/1. Un ejemplo de fichero escrito por Prolog sería (los saltos de linea se han introducido para facilitar la lectura y en realidad no pertenecerían al fichero):

```
progresion([ (cifrado(grado(i), matricula(maj7)), figura(2, 4))
           , (cifrado(grado(iii), matricula(m7)), figura(2, 4))
           , (cifrado(grado(vi), matricula(m7)), figura(2, 8))
           , (cifrado(grado(vii), matricula(m7b5)), figura(2, 8))
           , (cifrado(grado(vii), matricula(m7b5)), figura(2, 4))
           , (cifrado(grado(iii), matricula(m7)), figura(2, 4))
         ]).
```

3.6. Otros usos de éste módulo

La interfaz gráfica desarrollada en C++ permite al usuario un acceso cómodo y fácil a las diversas funcionalidades que ofrece este módulo, sin tener que lidiar con Prolog por consola. Así pués la interacción con el generador de acordes es sencilla y puede aumentarse la complejidad de la manipulación de las progresiones sin que esto le resulte complicado al usuario. Las funcionalidades que ofrece de este módulo conbinado con el interfaz gráfico son:

- Generación de progresiones de acordes: Se especifica el número de compases y el número de mutaciones a aplicar. Las mutaciones se han dividido en dos grupos, el primero integrado por cambia_acordes, aniade_acordes y quita_acordes, y el segundo por dominante_secundario y iim7. Se puede ser más fino en la especificación de las mutaciones e indicar el número de mutaciones de cada grupo a aplicar, o incluso de cada tipo concreto, para tener mayor control. En todos los casos el orden en que se aplican las mutaciones es aleatorio.
- Salvado y carga de las progresiones de acordes: Se pueden guardar las progresiones generadas para recuperarlas en futuros proyectos o emplearlas para distintas pistas o bloques de un proyecto. Apilar pistas de acompañamiento con la misma progresión de acordes pero con estas progresiones transformadas en música de diferentes maneras, o incluso con distintas fuentes de sonido, es una opción interesante para componer.
- Modificación de una progresión: A una progresión también se le pueden aplicar mutaciones después de ser creada o cargada en GENARO. Estas mutaciones se pueden especificar con la misma finura que en la genaración e incluso se puede especificar un acorde concreto para mutar. Es una forma de emplear a GENARO como asistente a la composición.

- Edición manual de una progresión: GENARO incorpora un editor de progresiones de acordes, para modificar una progresión generado por GENARO que es interesante pero falla en algún punto, para que el usuario introduzca una progresión que le gusta y GENARO le haga una melodía y un bajo, o para que GENARO se encargue de pasar la progresión a música concreta... otra vez GENARO como asistente a la composición.

3.7. Capacidad de ampliación

Este módulo es fácilmente ampliable, en parte por las características de Prolog. No hay clases ni interfaces a los que ceñirse, basta con ajustarse a la especificación del predicado es_progresion/1 para trabajar en su mismo idioma. Las vías más evidentes por las que se podría ampliar este módulo son:

1. Adición de nuevas mutaciones: basta con crear predicados que tengan como entrada un término que cumpla es_progresion/1 y como salida otro término que también lo haga. Las reglas de la armonía sugieren la introducción de nuevas mutaciones usando técnicas como disminuidos de paso, intercambio modal, dominantes sustitutos. La extensión del módulo por esta vía sería muy sencilla.

2. Adición de nuevas progresiones semilla: bastaría con añadir a la base de datos nuevas progresiones que cumplieran las dos condiciones antes mencionadas, y que tuvieran interés estético. Esta extensión sería inmediata.

3. Modulacion y enlace de progresiones: se trataría de construir progresiones pensadas para suceder a otras progresiones, siguiendo una lógica marcada por las leyes de la armonía. La dificultad aquí estaría en la cuestión teórica musical más que en la informática, puesto que la abstracción de las progresiones de acordes y su implementación suponen un acceso sencillo para enfrentarse a este problema.

4. Algo totalmente diferente: Cualquier programa en cualquier lenguaje que lea y escriba archivos de texto en el formato especificado por el predicado es_progresion/1 puede comunicarse con éste módulo y extenderlo o sustituirlo.

Capítulo 4

Traducción de cifrados

4.1. Introducción

La finalidad del módulo de traducción de cifrados hecho en Haskell es que proporcione todas las notas exactas que poseerá cada cifrado en función, por supuesto, de ciertas opciones de entrada y, en caso de ambigüedad, usará la aleatoriedad para elegir entre posibles resultados.

Cuando nos referimos a las notas exactas de un cifrado musical nos referimos a las alturas de las notas, tanto el *PitchClass* como la octava, es decir, aquello que determina la frecuencia del sonido. No nos referimos a cómo esas notas se van a ejecutar en el tiempo porque eso es parte de la siguiente etapa en la que vamos a encajar estas notas con lo que hemos llamado patron rítmico. Por ello no es necesario tener en cuenta la duración individual de cada nota, que sería lo único que nos faltaría para tener una verdadera nota musical, sino que basta tener la duración del acorde. Es por ello que la salida de este módulo es el tipo:

```
type AcordeOrdenado = ([Pitch],Dur)
```

es decir, la duración del acorde junto con las notas que tiene.

Pongamos un ejemplo. El cifrado de C Maj7 (do mayor con séptima mayor), posee las notas (C,E,G,B) pero esa información está incompleta para una traducción verdadera. Supongamos que buscamos que ese acorde tenga 7 notas, esté en estado fundamental (el C como nota más grave) y en primera disposición (el C como nota más aguda). Es obvio que tenemos que duplicar algunas alturas para que se cumpla lo anterior. El programa actuará devolviendo lo siguiente: (C,E,G,B,C,G,C). Ahora sí que se cumple lo anterior pero la traducción sigue siendo incompleta según lo dicho anteriormente ya que falta la octava. Digamos que ahora queremos que la nota más grave esté en la cuarta octava y las demás notas sean la siguiente menor más aguda. Entonces el acorde quedaría como sigue: ((C,4),(E,4),(G,4),(B,4),(C,5),(G,5),(C,6)) que sí es lo que buscamos.

La etapa de traducción se compone de dos fases:

1. Traducción a forma fundamental: simplemente indica los *PitchClass* que posee el acorde
2. Reparto de las notas en voces: duplica y permuta los *PitchClass* de la etapa 1 para que se ajusten a unos determinados parámetros. Vamos adelantando que los parámetros más importantes son los dos sistemas de traducción: el continuo y el paralelo.

4.2. Traducción a forma fundamental

4.2.1. Vectores de matrículas

Un cifrado se compone de dos partes: el grado sobre el que se asienta y la especie que es. Nosotros hemos llamado a la especie *matrícula*, de esa forma, por ejemplo, el cifrado de *C Maj7* posee de matrícula *Maj7* y de vector de *PitchClass* [C,E,G,B]; el cifrado de *C m* (do menor) posee matrícula *m* y *PitchClass* [C,bE,G].

Independientemente del grado del cifrado todos los acordes que son mayores posee una tercera mayor y una quinta justa (un músico entendería esto perfectamente aunque no es necesario para lo que nos trata a continuación). Una tercera mayor está compuesta de 4 semitonos (ó 2 tonos) y una quinta justa de 7 semitonos (ó 3 tonos y medio). De esa forma es fácil ver que para obtener un acorde mayor sobre cualquier nota simplemente tenemos que sumar 4 semitonos a dicha nota (obteniendo la tercera del acorde) y de nuevo 7 (para obtener la quinta). Más fácil es ver todavía que eso se puede expresar en forma de tupla como (0,4,7). Veamos un ejemplo para aclararlo:

Buscamos el acorde de D mayor. El *PitchClass* D en *Haskore* se representa por el número 2, cosa que nos facilita la suma. Si sumamos 2 + (0,4,7) nos da (2,6,9) que pasándolo otra vez a notas nos da (D,Fs,A) (Fs es Fa sostenido). Aquí hay que decir que tanto *Haskore* como MIDI usa un sistema atemperado, es decir, que las notas Fs y Gf (Sol bemol) representan el mismo sonido. Una vez entendido este concepto pasamos a mostrar todos los vectores asociados con las matrículas que maneja Genaro:
Mayor → [0,4,7]
Menor → [0,3,7]
Au → [0,4,8]
Dis → [0,3,6]
Sexta → [0,4,7,9]
Men6 → [0,3,7,9]
Men7B5 → [0,3,6,10]
Maj7 → [0,4,7,11]
Sept → [0,4,7,10]
Men7 → [0,3,7,10]
MenMaj7 → [0,3,7,11]
Au7 → [0,4,8,10]
Dis7 → [0,3,6,9]

Las funciones de Haskell que encapsulan esto se encuentran en el modulo *Progresiones.hs* y se llaman *matriculaAVector* y *sumaVector*.

4.2.2. Traducción a forma fundamental

Una vez que tenemos a nuestra disposición los vectores de las matrículas ya podemos traducir cualquier cifrado a su forma fundamental:

```
traduceCifrado :: Cifrado -> [PitchClass]
traduceCifrado (grado, matricula) = map  absPitchAPitchClass
        (sumaVector (matriculaAVector matricula) (gradoAInt grado))
```

4.3. Reparto de las voces

Existen dos formas principales de ordenar las notas de la etapa anterior: sistema paralelo y sistema continuo. Ambas formas han sido inspiradas del libro de Enric Herrera.

4.3.1. Sistema Paralelo

El sistema paralelo es el más simple. Básicamente se encarga de colocar todos los acordes en una inversión y una disposicion dada. Los parametros que necesitan que sean pasados por el usuario son los siguientes:

1. octava inicial: en la octava a partir de la cual se empiezan a colocar las notas.

2. inversión: un entero que indica la inversión que es.

3. disposición: otro entero que indica la disposición del acorde.

4. número de notas: número de notas final del acorde.

La inversión de un acorde es simplemente la indicación de qué nota es la más grave en un acorde. Una inversion de 1 indica que el acorde está en primera inversión (la tercera del acorde como nota más grave) y así. La inversión 0 se usa para decir que estamos en estado fundamental (la fundamental del acorde como nota más baja).

Algo parecido pasa con la disposición que únicamente depende de la nota más aguda. Una disposición de 1 indica que la fundamental del acorde es la nota más alta y así.

Tanto la inversión como la disposicion depende únicamente de la nota más grave y de la más aguda y eso implica dos cosas. Primera es que el número mínimo de notas que puede tener un acorde es dos (las dos notas extremas). Se podría haber hecho de otra forma pero esta es la que vimos más coherente con los parámetros. Segundo es que no hay información sobre la colocación del resto de las notas. Podríamos haber seguido una opción parecida a la del sistema continuo (como ya veremos) pero pensamos que iba más encaminado con la filosofía de este sistema (según Enric Herrera, por supuesto) que las notas intermedias se colocaran en el mismo orden que establece su estado fundamental.

Una nueva abstracción tenemos que introducir en este momento: el AcordeSimple.

```
type AcordeSimple = [Int]
```

Un acorde simple es solamente una lista de números. Cada número es un índice sobre la lista de notas del acorde que se pasa de la etapa anterior, es decir, sobre un acorde en estado fundamental. Para formar un acorde simple según los parametros del sistema paralelo usamos

```
formarAcordeSimple :: NumNotasFund -> NumNotasTotal -> Inversion
                      -> Disposicion -> AcordeSimple
formarAcordeSimple numNotasFund numNotasTotal inv disp =
        reverse (disp : anadeAcordeSimple numNotasFund (numNotasTotal - 2) [inv + 1])
```

¿Pará qué nos sirve esto? Simplemente independiza las notas verdaderas del acorde con su ordenación. Esto se entiendo muy fácilmente con un ejemplo:
Supongamos el acorde de C Mayor de nuevo cuyo estado fundamental es [C,E,G] que es proporcionado por la etapa 1. Queremos este acorde con 10 notas, en primera inversión, y en tercera disposición. Llamando a la función *formarAcordeSimple* tenemos:

```
TraduceCifrados> formarAcordeSimple 3 10 1 3
[2,3,1,2,3,1,2,3,1,3]
```

¿Qué significa esto? Significa que el primer elemento es el segundo elemento de [C,E,G], el segundo es el tercer elemento de [C,E,G] y así. La importancia de esto es que lo que hemos llamado acorde simple es independiente de las notas concretas a las que se refiere.

Ahora simplemente hay que fusionar las dos cosas pero eso es muy fácil ya que son índices (recordemos que con el 1 nosotros nos referimos al primer elemento, a diferencia de en Haskell que se refiere al primer elemento con el 0)

```
encajaAcordeSimple :: [PitchClass] -> AcordeSimple -> [PitchClass]
encajaAcordeSimple lpc [] = []
encajaAcordeSimple lpc (num : resto) = (lpc !! (num - 1)) : encajaAcordeSimple lpc resto
```

Ahora sólo nos queda la parte de añadir la octava.

4.3.2. Sistema Continuo

La idea principal de este sistema es que las voces de un acorde se muevan lo menos posible respecto a las voces del acorde anterior, de esa forma el efecto transitorio entre acordes será más suave.

Más detalladamente se busca lo siguiente

1. Si un acorde posee notas en común respecto del anterior entonces esas notas tienen que ser la misma
2. El resto de las voces se mueve a la nota más cercana.

Enric Herrera propone en la página 42 un método que nos hace saber cuantas notas en común poseen esos dos acordes. En primer lugar ese método es sólo para triadas aunque se podría ampliar sin mucha dificultad a cuatriadas. Por otro lado sólo tiene en cuenta los acordes diatónicos. Eso es un problema si queremos usar algo parecido sobre acordes no diatonicos como puede ser los dominantes secundarios. Por tanto este método no es suficientemente general para nuestros propositos y tenemos que ampliarlo.

El algoritmo se divide en cuatro partes.

1. Traducimos todos los cifrados a su estado funtamental sin duplicar ni permutar los PitchClass, es decir, lo que obtenemos directamente de usar el Vector de la matrícula.
2. Permutamos cada acorde hasta que la diferencia de las notas sea la más pequea con respecto al acorde anterior.

3. Duplicamos las notas para que posean el número de notas deseadas.

4. Insertamos la octava para que la altura de las notas este fijada definitivamente.

Ahora vamos a verlos en detalle.

Traducción a estado fundamental

La traducción a estado fundamental no merece demasiada atención ya que se realiza de la misma forma a la del sistema paralelo, usando la función *traduceCifrado*.

Obtención del acorde más cercano al anterior

Esta es, según mi criterio, la parte más importante del algoritmo del sistema continuo. La base del algoritmo es bastante fácil. En primer lugar necesitamos dos operaciones sobre listas de PitchClass: una que nos diga las notas en común en la misma posición de la lista y otra que nos diga la distancia entre dos listas de PitchClass. Esto nos da lo siguiente:

```
coincidencias :: [PitchClass] -> [PitchClass] -> Int
coincidencias lp1 lp2 = foldr1 (+) (map fromEnum [(lp1 !! i) ==
                        (lp2 !! i) | i <- [0..(longMen - 1 )]])
where longMen = min (length lp1) (length lp2)

distancia :: [PitchClass] -> [PitchClass] -> Int
distancia lp1 lp2 = foldr1 (+) [abs ( ((map pitchClass lp1) !! i) -
                    ((map pitchClass lp2) !! i) ) | i <- [0..(longMen - 1 )] ]
where longMen = min (length lp1) (length lp2)
```

Una vez que tenemos estas dos funciones ya podemos sacar el acorde más cercano a otro de la siguiente forma. Primero hacemos todas las permutaciones de la lista de PitchClass que representa el acorde en posición fundamental. Eso no es tan preocupante computacionalmente hablando ya que nosotros vamos a tener como máximo cuatriadas que forman listas de cuatro elementos (4‘ combinaciones). Segundo vamos a sacar los acordes cuya función *coincidencias* nos de el mayor valor y eliminar los otros de la lista. Tercero vamos a sacar los acordes (de los que quedan) cuya función distancia al anterior sea la menor. Cuarto, de los restantes vamos a elegir uno al azar. Muy posiblemente ya en la segunda etapa solo nos quede una lista de un elemento aunque hay que asegurarse hasta el final.

El primer acorde de la progresión es un caso especial ya que no tiene ningún acorde anterior con el que poder compararse. Lo que hacemos es simplemente elegir una inversión aleatoria para él.

El algoritmo completo se recoge en la función organiza que es:

```
organizar :: RandomGen a => a -> [[PitchClass]] -> [[PitchClass]]
organizar gen (x:xs) = elegido : organizarRec sigGen elegido xs
    where lista = [inversion i x | i <- [0..(length x - 1 )]];
          (elegido, sigGen) = elementoAleatorio gen lista

organizarRec :: RandomGen a => a -> [PitchClass] -> [[PitchClass]] -> [[PitchClass]]
organizarRec _ _ [] = []
organizarRec gen referencia (x:xs) = nuevaRef : organizarRec sigGen nuevaRef xs
    where (nuevaRef, sigGen) = masCoincidente gen referencia x
```

Donde inversion es la i-esima inversion de la lista x.

Duplicacion de notas

La idea es la de usar el llamado AcordeSimple para acceder a las notas ya ordenadas de los acordes. Ahora todos los acordes simples se van a forman empezando desde el número 1. Eso no implica que el acorde vaya a estar en forma fundamental ya que el índice 1 sobre la lista de PitchClass no corresponde a la fundamental del acorde porque la etapa anterior se ha encargado de permutar la lista adecuadamente. El código de éste algoritmo se encuentra integrado en al misma función que la del siguiente por lo que se verá después cómo es.

Inserción de octava

Ahora solamente falta insertar la octava en las listas de PitchClass para que tengan la altura correspondiente. Suponemos que el acorde anterior ya tiene todas las octavas añadidas y buscamos introducirlas al actual. No podemos hacer que empiecen a añadirse desde la más grave como se hizo en el sistema paralelo porque eso podría darnos alturas diferentes para notas que en teoría son iguales. Hay que hacer lo siguiente: buscar la primera nota coincidente entre los dos acordes y tomar esa octava, dividir el acorde actual por dos por dicha nota. Una vez dividido la parte más grave del acorde se le asigna octavas de forma descendente y la más aguda se hace de forma creciente. Como las notas comunes están en la misma posición dentro de las listas de PitchClass nos asegura que dichas notas comunes reciben la misma octava

```
traduceProgresionSistemaContinuo :: RandomGen a => a -> OctaveIni
        -> NumNotasTotal -> Progresion -> [AcordeOrdenado]
traduceProgresionSistemaContinuo gen octaveIni numNotasTotal progresion
= zip (arreglaTodos octaveIni (map2 encajaAcordeSimple
            (organizar gen (map traduceCifrado cifrados)) listaAcordesSimples))
            duraciones
where desabrochar = unzip progresion ;
cifrados = fst desabrochar ;
duraciones = snd desabrochar ;
listaAcordesSimples = map reverse [anadeAcordeSimple (matriculaAInt
                                        ((map snd cifrados)!!i)) (numNotasTotal-1) [1]
                                        | i <- [0..(length cifrados-1)] ]
```

Donde *arreglaTodos* se encarga de introducir la octava por el procedimiento ahora dicho y *organiza* traduce los cifrados por el sistema continuo.

4.4. Mejoras

Pocas objecciones tengo respecto de este módulo. Quizás lo que menos me ha gustado ha sido la forma en que se introduce la octava inicial. En el sistema paralelo no veo que tenga más remedio pero en el sistema continuo habría sido bueno implementar un método para que las notas de los acordes se mantuviera entre ciertos limites definidos por el usuario. El método actual puede hacer que las notas de

los acordes suban o bajen más y más de un acorde al siguiente llegando incluso a pasarse de las octavas permitidas si introducimos gran cantidad de acordes. Por suerte esto no pasará debido a la función *eliminaOctavasErroneas* que deja las octava menores de 0 a 0 y las mayores de 21 a 21 (0 y 21 son los límites del estandar MIDI). No es lo mejor pero, por lo menos, no da error.

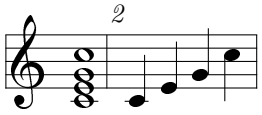
Capítulo 5

Patrones Rítmicos

5.1. Introducción

La etapa del patrón rítmico es la que se ejecuta posteriormente a la de traducción de los cifrados. Es por ello que lo que recibe esta etapa es una lista de lo que hemos llamado "acordes ordenados" (el tipo *[AcordeOrdenado]*). La idea principal de esta parte es repartir las voces de los acordes ordenados en el tiempo. Es por ello que vamos a buscar una estructura que sea capaz de organizar las voces en el tiempo, que sirva para cualquier acorde independientemente del número de voces o duración que posea y qué se pueda guardar para posteriores usos.

Para ilustar estas ideas veamos un ejemplo simple. Supongamos que de la etapa anterior de traducción de cifrados nos llega lo siguiente [([(C,4),(E,4),(G,4),(C,4)],1



Aunque las notas son las mismas ellas estan dispersas en el tiempo de diferente forma.

5.2. Tipo y Funcionamiento

El tipo del patrón rítmico es el siguiente:

```
type Voz = Int      -- Voz: la primera voz, en este caso, es la mas grave
type Acento = Float -- Acento: intensidad con que se ejecuta
                    --         esa nota. Valor de 0 a 100
type Ligado = Bool  -- Ligado: indica si una voz de una columna esta
                    --         ligada con la se la siguiente columna.
type URV = [(Voz, Acento, Ligado)] -- Unidad de ritmo vertical, especifica todas
                                   -- las filas de una unica columna

type URH = Dur  -- Unidad de ritmo horizontal, especifica
                                   -- la duracion de una columna
```

```
type UPR = ( URV , URH)             -- Unidad del patron ritmico, especifica completamente
                                    -- toda la informacion necesaria para una columna
                                    -- (con todas sus filas) en la matriz que representa el patron

type AlturaPatron = Int -- numero maximo de voces que posee el patron

type MatrizRitmica = [UPR]          -- Una lista de columnas, vamos, como una matriz

type PatronRitmico = (AlturaPatron, MatrizRitmica)
```

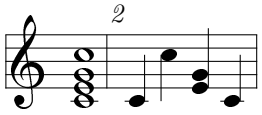
La idea más simple de ver un patrón rítmico es como una matriz (aunque no lo sea exactamente ya que el número de filas por cada columna varía) que representa una plantilla con agujeros. Su altura (número de filas) es el número de voces y la anchura (número de columnas) una duración en forma de fracción de Haskore. Dicha plantilla se encaja encima del acorde ordenado; aquellas voces que caigan dentro de un agujero se ejecutarán en ese instante de tiempo. Por ejemplo, supongamos la siguiente plantilla con agujeros que representa el arpegio anterior de cuatro voces:

```
- - - X
- - X -
- X - -
X - - -
```

donde las X representan los agujeros. Ahora supongamos que la plantilla es esta:

```
- X - -
- - X -
- - X -
X - - X
```

entonces para el siguiente acorde tendriamos lo siguiente cuando aplicáramos el patrón:



es decir, primero la voz 1 (recordemos que para facilitarme el trabajo ahora las voces se numeran de abajo a arriba), luego la voz 4, posteriormente la 2 y la 3 a la vez y, por último, la 1 otra vez.

Esta forma de ver las cosas es útil porque el editor de patrones rítmicos está implementado para que se parezca a esto aunque el tipo *PatronRitmico* de Haskell necesita más información para completarse. Por ejemplo, el último patrón rítmico en Haskell se escribiría así :

```
patronEj1 :: PatronRitmico
patronEj1 = ( 4, [( [(1,100,False)] , 1%4),
                 ( [(4,100,False)] , 1%4),
                 ( [(2,100,False),(3,100,False)] , 1%4),
                 ( [(1,100,False)] , 1%4)] )
```

5.3. Encaje del Patrón Rítmico

Ahora vamos a ver como fusionar un patrón rítmico con un acorde ordenado para que se forme la salida definitiva de esta etapa, que es un tipo Music de Haskore y ya representa música (e incluso se puede pasar a midi).

5.3.1. Encaje perfecto

Primeramente vamos a ver como encajar un patrón cuando el número de voces, tanto del patrón como del acorde ordenado, es el mismo y cuando la duración de ambos también es la misma. De esa forma todo encaja perfectamente.

El encaje se hace en dos partes. En una primera parte, la que corresponde más claramente con un encaje, se sustituye cada emphVoz dentro del patrón rítmico por un emphPitch.

```
encaja :: [Pitch] -> URV -> [(Pitch, Ligado, Acento)]
encaja lp [] = []
encaja lp ( (voz, acento, ligado) : resto) =
      ( lp !! (voz-1), ligado, acento) : encaja lp resto
```

Posteriormente se introduce en el Pitch la duración y el Acento formando una emphNote de Haskore.

```
insertaDur :: Dur -> [(Pitch, Ligado, Acento)] -> [(Music, Ligado)]
insertaDur dur [] = [(Rest dur, False)]
insertaDur dur lp = insertaDur2 dur lp

insertaDur2 :: Dur -> [(Pitch, Ligado, Acento)] -> [(Music, Ligado)]
insertaDur2 dur [(pitch, ligado, acento)] = [ ( Note pitch dur [Volume acento] , ligado ) ]
insertaDur2 dur ((pitch, ligado, acento) : resto) =
      ( Note pitch dur [Volume acento] , ligado ) : insertaDur2 dur resto
```

De esa forma obtenemos una estructura que hemos llamado emphNotasLigadas.

```
type NotasLigadasVertical = [(Music,Ligado)]
type NotasLigadas = [(NotasLigadasVertical,Dur)]
-- Donde Dur es la duracion del acorde
```

La razón que fundamenta el uso de esta estructura es, simplemente, que Haskore no nos da la opción, dentro de sus constructoras del tipo Music, de hacer que una *Note* se ligue con la siguiente *Note*. Es por eso que necesitamos un paso previo antes del tipo Music que se espera que se devuelva en esta etapa. Dicho paso puede ser perfectamente transparente a la persona que use este módulo.

El algoritmo para resolver lo anterior es difícil de implementar aunque fácil de entender. Dada la lista que hemos llamado NotasLigadas se comienza por el final de dicha lista y se avanza hacia atrás. Si la nota actual lleva en la tupla un *True* en el tipo *Ligado* entonces se ligara (si se puede) con una de las notas siguientes. Por empezar desde el final de la lista cada paso de ligado es correcto y cuando lleguemos a la cabeza la lista completa estara arreglada.

```
eliminaLigaduras :: NotasLigadas -> NotasLigadas
eliminaLigaduras [n] = [n]
eliminaLigaduras ((notasVerticales, dur) : resto) =
  let { eliminadas = eliminaLigaduras resto ;
        cabeza = head eliminadas ;
        arregladoCabeza = buscaTodasNotas notasVerticales (fst cabeza)
  } in (fst arregladoCabeza, dur) : (snd arregladoCabeza, snd cabeza) : tail eliminadas

buscaTodasNotas :: NotasLigadasVertical -> NotasLigadasVertical
                   -> ( NotasLigadasVertical , NotasLigadasVertical )
buscaTodasNotas notas1 notas2 = buscaTodasNotas2 notas1 [] notas2

buscaTodasNotas2 :: NotasLigadasVertical -> NotasLigadasVertical -> NotasLigadasVertical
                    -> (NotasLigadasVertical, NotasLigadasVertical)
buscaTodasNotas2 [] notas1 notas2 = ( notas1, notas2 )
buscaTodasNotas2 (( nota, False ) : restoPitch ) notas1 notas2 =
      buscaTodasNotas2 restoPitch ((nota,False):notas1) notas2
buscaTodasNotas2 (( Note pitch dur lA, True ) : restoPitch ) notas1 notas2 =
buscaTodasNotas2 restoPitch ((Note pitch (dur + buscaNota pitch notas2) lA, False):notas1)
      (eliminaNota pitch notas2)
\end{vebatim}
\normalsize

Una vez que hemos arreglado todas las notas respecto a sus ligaduras solamente nos queda
pasar la matrix \emph{NotasLigadas} a un tipo Music secuenciando las columnas y paralelizando
las filas.

\small
\begin{verbatim}
-- deNotasLigadasAMusica: dada la lista de notas ligadas (ya sea bien arregladas o no)
-- las pasa a musica Haskore
deNotasLigadasAMusica :: NotasLigadas -> Music
deNotasLigadasAMusica = deNotasLigadasAMusica2 (0%1)

-- deNotasLigadasAMusica2: es la funcion recursiva de deNotasLigadasAMusica y con acumulador.
-- El parametro dur indica la duracion que hay que dejar hasta el comiento
-- de la cancion antes de interpretar las notas ligadas a tratar
deNotasLigadasAMusica2 :: Dur -> NotasLigadas -> Music
deNotasLigadasAMusica2 dur [(nV,_)] = Rest dur :+: paraleliza nV
deNotasLigadasAMusica2 dur ((nV,dur2):resto) = (Rest dur :+: paraleliza nV) :=:
                                               deNotasLigadasAMusica2 (dur + dur2) resto

-- paraleliza: ejecuta en paralelo a lista de musica sin intereserse por el parametro booleano
paraleliza :: [( Music, Bool )] -> Music
paraleliza [] = Rest (0%1)
paraleliza [ ( nota, _ ) ] = nota
paraleliza (( nota, _ ):resto) = nota :=: paraleliza resto
```

Como Haskore no posee un Music vacío usamos *Rest (0 % 1)* para referirnos a algo inútil musicalmente hablando.

5.3.2. Problemas de encaje

En la sección anterior hemos hablado de cómo es el encaje del patrón cuando el número de voces y la duración de tanto el patrón como del acorde ordenado eran los mismos. Pero en la mayoría de los casos no va a ser así. Generalmente el acorde ordenado tendrá diferente duración a la del patrón y muy posiblemente el número de voces de ambos será diferente. Sería un error crear un nuevo patrón para cada acorde con sus pecularidades así que hemos introducido opciones para solucionar eso.

Problemas de encaje horizontal

Se produce cuando la duración del patrón es diferente a la duración del acorde ordenado al que se quiere encajar.

Si la duración del patrón es menor entonces el problema no es muy importante ya que vamos a repetir dicho patrón tantas veces como haga falta hasta rellenar el acorde completamente o hasta que nos sobre algo, en cuyo caso pasaremos a la siguiente opción.

Si la duración del patrón es mayor que la del acorde o, dicho de otra forma, sobra algo del patrón cuando lo encajamos en el acorde ordenado entonces la pregunta es ¿qué hacemos con el trozo que sobra? A nosotros se nos han ocurrido dos cosas:
La primera es que ese trozo se pierda comenzando el siguiente acorde con, otra vez, el principio del patrón rítmico.
La segunda es que dicho trozo sea el comienzo del siguiente acorde, es decir, que se le pase al siguiente acorde.
La primera opción la hemos llamado encaje horizontal no cíclico y a la segunda encaje horizontal cíclico. Veamos un ejemplo para asentar ideas:

Sea el patrón rítmico del arpegio de cuatro voces que hemos visto antes

```
- - - x
- - x -
- x - -
x - - -
```

Ahora veamos el caso no cíclico

```
 - - - x - - - - - - - - - x - - - - - - x - - - x - -
 - - x - - - x - - x - - x - - - - - - - x - - - x - - -
 - x - - - x - - x - - x - - - - - x - x - - - x - - - x
 x - - - x - - x - - x - - - - x x - x - - - x - - - x -
|-------------|-----|---------|---|-------------------|   duracion acorde
```

Recordemos que la misma voz en diferentes acordes no tiene porque corresponder a la misma altura de nota.
Ahora veamos el caso cíclico para la misma sucesión de acordes

```
 - - - x - - - x - - - x - - - x - - - x - - - x - - -
 - - x - - - x - - - x - - - x - - - x - - - x - - - x
 - x - - - x - - - x - - - x - - - x - - - x - - - x -
 x - - - x - - - x - - - x - - - x - - - x - - - x - -
|-------------|-----|---------|---|--------------------|   duracion acorde
```

problemas de encaje vertical

Algo parecido ocurre cuando el número de voces del acorde y el del patrón es diferente pero en este caso tenemos más opciones. Veamos los diferentes casos.
Si la altura del acorde es mayor que la del patrón:
1. Truncar: las voces extra del acorde se ignoran

```
         -
 - - - - |
 - - - - |
 - - - x | altura del acorde
 - - x - |
 - x - - |
 x - - - |
         -
```

2. Saturar: supongamos que el acorde tiene A voces y el patrón tiene P. En este caso las voces desde P+1 hasta A se tocan a la vez cuando el patron toca su voz P-esima.

```
         -
 - - - x |
 - - - x |
 - - - x | <- P-esima voz del patron
 - - x - |
 - x - - |
 x - - - | altura del acorde
         -
```

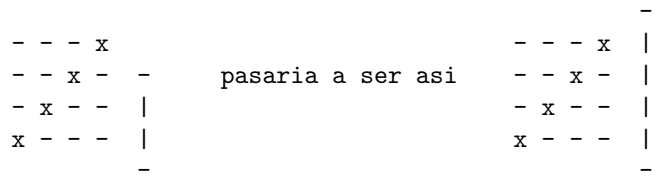
Si la altura del acorde es menor que la del patrón:
1. Truncar (no cunfundir con el truncar anterior ya que son casos distintos aunque tengan el mismo nombre): Las voces desde A+1 hasta P no se tocan.

```
          -
 - x - -  |
 x - - -  | altura del acorde (solo 2 voces)
          -
```

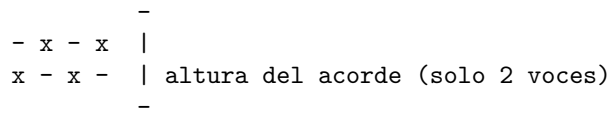
2. Saturar (no cunfundir con el saturar anterior ya que son casos distintos aunque tengan el mismo nombre): Las voces desde A+1 hasta P no se tocan se toman como la voz A-esima del acorde.

```
          -
 - x x x  |
 x - - -  | altura del acorde (solo 2 voces)
          -
```

3. Cíclico: En este caso lo que se cambia es el acorde para que tenga el mismo número de voces que el patrón. Ello se consigue repitiendo las notas del acorde pero aumentando la octava de forma que el acorde siga ordenado en altura.



4. Módulo: Si una voz X del patrón está en el rango A+1 y P, es decir, que cae fuera del rango del patrón, entonces esa voz se transforma en X := ((X-1) mod A) + 1. Eso nos asegura que X cae entre 1 y A. A-esima del acorde.



Los detalles de implementación no merecen la pena ponerlos aquí ya que son muy simples. Cabe señalar que simplemente hay que cambiar lo que nosotros hemos llamado unidad de ritmo vertical (*URV=[(Voz,Acento,Ligado)]*) añadiendo o quitando voces para que se ajuste a las voces del acorde.

5.4. Mejoras

5.4.1. Patrones rítmicos dinámicos

Lo que hemos dicho hasta aquí podría definirse como el uso de patrones rítmicos estáticos, es decir, que no se generan en tiempo de llamada. Una posible mejora sería introducir patrones rítmicos dinámicos, que dado algunos parámetros te generara un patrón rítmico. Obviamente esto sólo se podría hacer para patrones simples y que cumpliesen alguna propiedad escalable. El ejemplo más claro es el del arpegio o el que toca todas las notas a la vez. Para el arpegio dinámico podríamos hacer una función que, dado una duración para las notas y un número de notas, devolviera un patrón rítmico correspondiente con un arpegio de esas características:

```
arpegioDinamico :: Dur -> Int -> PatronRitmico
arpegioDinamico d n = ( n, [ ([(i,100,False)] , d) | i<-[1..n] ] )
```

Algo tan simple como eso daría mucho juego y flexibilidad al programa.

5.4.2. Patrones aleatorios

Sería algo así como un patrón dinámico pero la matriz sería generada aleatoriamente. Quizás introducir conceptos (posiblemente el ritmo) que restringieran esa aleatoriedad podría hacer más audible un patrón de estas características.

Capítulo 6

Generador de melodías

6.1. Introducción

El módulo generador de melodías se encarga de para componer una melodía a partir de una progresión de acordes y un patrón rítmico. Se utilizan las reglas de la armonía para que la melodía resultado este relacionada con la progresión y 'quede bien' al sonar a la vez que la traducción a música de ésta.
Todas las melodías compuestas por GENARO son monofónicas, esto es, son sucesiones de notas en el tiempo tales que no suena más de una nota a la vez. Las notas no se superponen en el tiempo como en el caso de los acompañamientos. Esta simplificación facilita mucho el proceso de composición.

6.2. Abstracciones empleadas

La generación de melodías de GENARO se basa en dos abstracciones, una *curva melódica* y una *lista de acentos*:

1. La curva melódica: es una lista de enteros que se entienden como saltos dentro de una escala que no se ha fijado todavía.
    - Un salto de 0 indica quedarse en la misma nota, un salto de n con n distinto de 0 indica moverse n notas dentro de una escala, hacia arriba en el caso de que n sea positivo y hacia abajo si es negativo.
    - Una curva melódica se refiere por tanto a las alturas de las notas solamente, no indica nada referido a duraciones ni al ritmo.
    - Una curva melódica representa las relaciones de altura relativa entre las notas que conforman la melodía, y en ningún momento se refiere a alturas absolutas. En un paso determinado del proceso de generación de melodías se realiza la correspondencia entre una curva melódica y una lista de alturas, especificando una escala y una altura de partida que fija una referencia absoluta desde la que comenzar el proceso.
    - Al pasar a la lista de alturas es posible que la altura de partida no pertenezca a la escala. En ese caso se entenderá que en el primer punto de la curva saltamos a una altura de la escala.

2. La lista de acentos: es una lista de parejas (float, fraccion entera) que expresa un perfil rítmico de la melodía.

   - Cada elemento de la lista de acentos representa la duración de una posible nota (segunda componente) y la fuerza con la que ésta se atacaría por parte del instrumentista que interpretará la melodía (primera componente):
     (a) Las duraciones se miden como figuras musicales.
     (b) La intensidad con que se ataca cada nota se mide de 0 a 100. Los casos de intensidades que valen 0 corresponden a los silencios, componentes fundamentales de la música.
   - Las listas de acentos se construyen de forma determinista procesando patrones rítmicos pero su aplicación no es en absoluto determinista, como se verá más adelante. Se pueden entender como la correspondencia de una dimensión de los patrones rítmicos, que tienen dos dimensiones.

Otra abstracción menos original pero también importante es el concepto de *registro*. Un registro es una restricción sobre las alturas de las notas, que deben pertenecer a un intervalo determinado por éste. Asignamos un registro agudo a la melodía para que resalte sobre el resto de pistas. El bajo a su vez tendrá asociado un registro grave, como es lógico.

6.3. Algoritmo de generación de melodías

Utiliza las abstracciones anteriores. Para saber obtener una melodía para una progresión primero debemos aprender a obtener una melodía para un solo acorde, y luego a enlazar las melodías de cada acorde.
El algoritmo que hace una melodía para un solo acorde recibe de entrada una curva melódica, una lista de acentos, el acorde sobre el que tiene que construir la melodía, la altura desde la que empezar la melodía y una serie de parámetros enteros cuya utilidad explicaremos después. Partiendo esa información aplicamos los siguientes pasos:

1. Ajustar curva melódica con lista de acentos: se modifica la lista de acentos para que tenga la misma longitud que la curva aleatoria. Para ello se dividen en dos o se unen parejas de acentos, según la lista de acentos sea más corta o más larga que la curva melódica. Los acento que se dividen o fusionan se eligen aleatoriamente.

2. Elegir los acentos: se elige de forma aleatoria una sublista de tamaño aleatorio de la lista de acentos, dando mayor probabilidad de ser elegidos a los acentos más fuertes. También se guardan en otra lista los acentos rechazados.

3. Elegir los puntos de la curva melódica: se hace la correspondencia entre la curva melódica y una lista de alturas de nota empleando la escala correspondiente al acorde, la altura de partida, y el registro correspondiente a la melodía. Estas alturas concretas ya pueden ser valoradas como más o menos estables en el contexto armónico que establece el acorde. Se elige una sublista aleatoria de esta lista de alturas, de la misma longitud que la sublista de acentos anterior, dando mayor probabilidad de ser elegidas a las alturas más estables.

4. Construir la primera versión de la melodía: a partir de las tres sublistas anteriores se construyen dos listas, una a partir de los acentos y alturas elegidos, la lista de notas que suenan; y otra a partir

de la lista de acentos rechazados, la lista de silencios o notas que no suenan. Estas dos listas se mezclan siguiendo el orden de los acentos en la lista de acentos original, y se obtiene una primera versión de la melodía.

5. Mutar la melodía: el resto del proceso consiste en aplicar mutaciones sucesivas a la primera versión de la melodía. Estas mutaciones se aplican siempre en el mismo orden y según indican los 4 enteros de entrada al algoritmo.

   i) Alargar notas: considera todas las notas de la melodía que no sean silencios y a las que siga inmediatamente un silencio. Elige aleatoriamente un número aleatorio de estas notas y alarga su duración con la del silencio que les seguía, eliminándolo. Se aplica tantas veces como indique el primer entero. Da mayor probabilidad de ser alargadas a las notas con alturas más estables en el acorde.

   ii) Dividir notas: considera todas las notas de la melodía que no sean silencios y a las que siga una nota que no sea un silencio, aunque no sea inmediatamente (aunque haya silencios entre estas dos notas no silencio). Elije aleatoriamente una sola de estas notas y elige aleatoriamente una altura entre la de la nota elegida y la de la primera nota no silencio que le sigue en la melodía. Después elimina los silencios entre estas dos notas e en su lugar inserta una nueva nota y silencios de forma que la nueva nota tiene la altura elegida anteriormente y está a la misma distancia en el tiempo de las dos notas. En resumen, calcula la distancia en el tiempo entre dos notas e introduce una nota entre ellas, en la mitad de la distancia. Se aplica tantas veces como indique el segundo entero. Da mayor probabilidad de ser divididas a las notas más largas.

   iii) Dividir notas fino: es como dividir notas solo que en vez de dividir por dos divide por una potencia de dos. Se aplica tantas veces como indique el tercer entero, y en cada aplicación divide por una potencia de dos de exponente elegido aleatoriamente entre 0 y el cuarto entero. Da la misma probabilidad de ser elegidas a todas las notas.

Una vez ya sabemos construir una melodía para un acorde construir una melodía para una progersión es sencillo. El algoritmo recibe como entrada una progresión, un patrón ritmico y una serie de parámetros enteros entre los que se incluyen los anteriores. Partiendo esa información aplicamos los siguientes pasos:

1. Generar la curva melódica: GENARO tiene un algoritmo de generación aleatoria de curvas melódicas. Se le especifica el número de puntos deseado, el salto máximo (valor absoluto máximo de un punto de la curva) y un peso para el cambio de dirección de la curva (la curva cambia de dirección cuando pasa de hacer saltos positivos a hacer saltos negativos o vicecersa). El algoritmo genera una curva que cumple esas restricciones, y para ello parte de un primer punto que vale 0, y de una dirección de movimiento (arriba o abajo) elegida aleatoriamente. Para el resto de puntos elige aleatoriamente si cambia de dirección o no teniendo en cuenta el peso suministrado y el tiempo que lleva 'caminando' en una dirección, cuanto más rato lleve mayor es la probabilidad de que cambie de dirección con un salto mayor. Si no cambia de dirección son más probables los saltos pequeños. Según estas probabilidades e intervalos de valores válidos va generando aleatoriamente los puntos de la curva.

2. Distribuir la curva entre los acordes: La curva melódica se genera para toda la progresión por lo que debe distribuirse entre los acordes de la progresión. A cada acorde le corresponde una cantidad de puntos de la curva según su duración, dando más puntos a los acordes más largos. A cada acorde le

corresponde un porcentaje del número de puntos de la curva igual a su duración entre la duración total de la progresión. Como esos porcentajes multiplicados por el número de puntos de la curva no suelen dar lugar a enteros se distribuye la parte entera y lo que sobra se reparte resolviendo aleatoriamente los conflictos. Una vez decidido cuantos puntos corresponden a cada acorde estos se reparten en orden, es decir, que si a cada acorde i le corresponden n_i puntos, entonces al acorde 1 le corresponden los puntos del 1 al n_i, al acorde 2 del $n_i + 1$ al 2 * (n_i), etc...

3. Construir la lista de acentos a partir del patrón rítmico: Esto se hace de forma determinísta, primero se calcula la lista formada por la media de los acentos de cada columna. Luego se calcula la media de estos acentos medios. Por último se procesa la lista de acentos medios de forma que los acentos mayores o iguales que la media se dejan igual, y los menores que la media se dejan a cero. Finalmente se empareja esta última lista con las duraciones correspondientes a cada columna. Esta lista de acentos será la misma para todos las melodías de todos los acordes de la progresión, de la misma forma que un patrón rítmico es común a todos los acordes de una progresión.

4. Construir las melodías para cada acorde: Primero se genera aleatoriamente una altura de partida para el primer acorde, dando una probabilidad mayor de ser elegidas a las notas mas estables de éste. Con esta altura de partida, la curva melódica que le corresponde, la lista de acentos, el primer acorde y los parámetros correspondientes se genera la melodía para el primer acorde. Las melodías de los acordes siguientes se construyen igual, pero tomando como nota de partida la última nota de la melodía del acorde anterior.

5. Componer las melodías: Las melodías resultado se componen secuencialmente.

6.4. Implementación

El módulo generador de melodías esta desarrollado íntegramente en Haskell, y se apoya en la librería Haskore [3] para componer melodías sobre un acompañamiento.
La comunicación de este módulo con el generador de progresiones se realiza a través de los ficheros de texto que escribe el generador de progresiones. Estos ficheros son leidos por Haskell y procesados por un parser generado a partir de la librería [5], para obtener elementos de tipo Progresión, un tipo definido en Haskell como sigue:

```
data Grado = I|BII|II|BIII|III|IV|BV|V|AUV|VI|BVII|VII|VIII
             |BBII|BBIII|AUII|BIV|AUIII|AUIV|BBVI|BVI|AUVI|BVIII|AUVIII
             |BIX|IX|BX|X|BXI|XI|AUXI|BXII|XII|AUXII|BXIII|XIII|AUXIII
             |V7 Grado
             |IIM7 Grado
     deriving(Show,Ord)

instance Eq Grado where
  g1 == g2 = (gradoAInt g1::Int) == (gradoAInt g2::Int)

data Matricula = Mayor|Menor|Au|Dis|Sexta|Men6|Men7B5|Maj7|Sept|Men7
                |MenMaj7|Au7|Dis7
     deriving(Enum,Read,Show,Eq,Ord,Bounded)
```

```
type Cifrado = (Grado, Matricula)

type Progresion = [(Cifrado, Dur)]
```

También se recurre a la librería [5] para leer los ficheros de patrón rítmico escritos por el editor de C ++.

El resto de manipulaciones se realizan utilizando los tipos de Haskore, las abstracciones anteriormente introducidas, y otros tipos que representan información sobre escalas y otra serie de información sobre teoría musical. Todos estos conceptos están representados en Haskell como sigue:

```
type SaltoMelodico = Int
type CurvaMelodica = [SaltoMelodico]
type CurvaMelodicaProgresion = [CurvaMelodica]
type ListaAcentos = [(Acento, Dur)]
type Acento = Float
{-
Identificadores de cada tipo de escala
-}
data Escala = Jonica|Dorica|Frigia|Lidia|Mixolidia|Eolia|Locria
              |MixolidiaB13|MixolidiaB9B13AU9|Disminuida
     deriving(Enum,Read,Show,Eq,Ord,Bounded)
{-
InfoEscala = (tensiones, lista de notas que la forman, notas a evitar)
-}
type InfoEscala = ([Grado], [Grado], [Grado])
{-
Indica cuales son las octavas apropiadas para cada instrumento
-}
type Registro = [Octave]
```

Las melodías como producto último se representan con la constructora Music de Haskore. Pero la entrada a las funciones de mutación es una lista de elementos de tipo Music, que representan cada uno notas individuales. Esta representación intermedía es más fácil de manipular, y se transforma fácilmente a la representación final, aplicando la operación de composición secuencial de Haskore.

6.5. Otros usos de éste módulo

La interfaz gráfica desarrollada en C++ ofrece al usuario un acceso sencillo a funcionalidades más sofisticadas. Las funcionalidades que ofrece de este módulo conbinado con el interfaz gráfico son:

- Manipulación de curvas melódicas: Se pueden manipular las curvas melódicas de diversas maneras:

  (a) Salvado y carga de curvas melódicas. Esto permite aplicar la misma curva melódica en diversas pistas y bloques de la composición. Como la aplicación de las curvas melódicas no es determinista, esto se puede entender como hacer variaciones sobre un tema común.

(b) Existe un editor manual de curvas melódicas, con el que el usuario puede diseñar curvas melódicas, visualizándolas como una linea de puntos en dos dimensiones, lo que facilita la comprensión intuitiva de su significado. Para que el usuario comprenda lo que hace al diseñar una curva, y lo que significan las curvas que hace GENARO.

(c) Existe una función que muta aleatoriamente curvas melódicas, cambiando tantos puntos como se le indique, dentro de un intervalo de variación determinado.

- Copia de melodías: Se puede copiar exactamente una melodía de una pista y subbloque determinado en otra pista y subbloque concretos. Esto es útil si se quiere repetir una melodía para rearmonizarla.
- Módulo de bajo: se basa fuertemente en este módulo, tanto en funciones a las que llama como en las ideas de sus algoritmos.

6.6. Capacidad de ampliación

La ausencia de objetos e interfaces y las abstracciones definidas favorecen la ampliabilidad de este módulo, proporcionando puntos de acceso para la generación de melodías. Las vías más evidentes por las que se podría ampliar este módulo son:

1. Diseñar nuevas mutaciones para la melodía. O permitir que se apliquen en distinto orden, o permitir mutar las melodías después de ser creadas.
2. Implementar un editor manual de melodías.
3. Si se amplía el módulo generador de acordes para que trabaje con nuevos tipos de acordes se debería ampliar también este módulo para que trate bien esos nuevos acordes. Por la estructura de éste esta ampliación sería muy sencilla.
4. Extrapolación de las curvas melódicas a partir de una melodía renderizada a midi.
5. Algo totalmente diferente: Cualquier programa en cualquier lenguaje que lea archivos de texto en el formato especificado por el predicado es_progresion/1, y ficheros de patrón rítmico puede sustituir a este módulo.

Quizás la aplicacion de la curva melodica es demasiado poco determinista como para hablar de variaciones al aplicarla varias veces o mutarla. Un concepto menos abstracto intermedio podría ser más apropiado. Ese concepto podría ser simplemente las listas de notas representadas por Haskore que son la entrada de los procedimientos de mutación. Si se implementara el acceso por interfaz a la mutación de melodías podríamos hablar con más justificación de variaciones sobre una melodía.
Y otra ampliación que sería muy interesante es introducir simetrías y estructuras como la frase cuadrada (motivo-contramotivo-motivo-resolución) en la generaciíon de melodías. GENARO produciría melodías muy pequeñas de las que luego daría más coherencia y lógica ajustandolas a estas estructuras. Para ello se podrían utilizar las mismas ideas de variaciones sobre una melodía comentadas en el párrafo anterior.

Capítulo 7

Armonización de Melodía

7.1. Introducción

Armonizar una melodía consiste, musicalmente hablando, en buscar acordes que "vayan bien" a una cierta melodía dada. Como veremos, la frase "vayan bien" simplemente es que cada nota de la melodía que vamos a armonizar sea parte de las notas que forman el acorde.
Por tanto, la entrada de este módulo es una melodía (puesta en forma de *Music* secuencial) y la salida es una progresión que lo armoniza. Obviamente el usuario va a poder seleccionar diferentes criterios para dicho fin.

El algoritmo completo se compone de dos partes. En una primera parte buscamos aquellas notas que vamos a querer que sean armonizadas. A esas notas las vamos a llamar *notas principales* y son aquellas que se pueden identificar más en un contexto vertical. En la segunda vamos a dar un acorde a una o varias notas principales. La concatenación de todos los acordes que nos van saliendo nos proporciona la progresión de salida de este módulo.

Antes de cualquier etapa hay que comprobar que el tipo *Music* que nos pasan es realmente una melodía, es decir, un *Music* exclusivamente secuencial (que sólo posea los operadores *:+:*, *Note* y *Rest*) y que además lo transforme en una estructura más manejable:

```
type Melodia = [ HaskoreSimple ]
data HaskoreSimple = Nota Pitch Dur
                   | Silencio Dur
     deriving(Eq, Ord, Show, Read)

deHaskoreSecuencialAMelodia :: Music -> Melodia
deHaskoreSecuencialAMelodia (Note p d _) = [ Nota p d ]
deHaskoreSecuencialAMelodia (Rest d)     = [ Silencio d ]
deHaskoreSecuencialAMelodia (m1:+:m2)    = deHaskoreSecuencialAMelodia m1 ++
                                           deHaskoreSecuencialAMelodia m2
deHaskoreSecuencialAMelodia _            = error "No es un Music solo secuencial"
```

7.2. Búsqueda de notas principales

Una vez que tenemos transformado la melodía (un *Music* secuencial) a una lista de notas y silencios (tipo *Melodia*) tenemos que seleccionar exclusivamente las notas que vamos a armonizar (notas princi-

pales) asegurándonos de que la suma de las duraciones de las notas principales sea la misma que la de la melodía pasada (para que los acordes duren lo mismo que la melodía).

¿Qué hacer con las notas que no son principales? Tanto las notas que no son principales como los silencios tiene que ser borrados por no considerarse parte de la armonía, sin embargo queremos que su duración no se pierda para que la longitud temporal de la melodía sea la misma que la progresión de acordes. Por consiguiente tienen que ceder su duración a la nota principal que tengan más a la izquierda.

Existe un caso especial: si la primera nota es no principal (es secundaria) no tiene ninguna nota principal a su izquierda. En ese caso cedera su duración a la que está a su derecha.
Como con todo en esta vida existen varias formas de hacerlo. Nosotros hemos seleccionado dos:

7.2.1. Notas de larga duración

Consideramos que se van a identificar más en un contexto armónico aquellas notas que tengan un duración larga. Pero, sin embargo, no podemos decir cómo de larga tiene que ser esa duración, sino que dependerá de la canción y, por tanto, será una apreciación subjetiva. Es por ello que tiene que ser pasada como parámetro. Por tanto, llegamos a lo siguiente: todas aquellas notas cuya duración sea mayor o igual a la duración mínima pasada como parámetro serán consideradas como notas largas. Lo habitual es que dicha duración mínima sea 1/4, que corresponde a una negra, aunque pueden darse otros casos.

```
deMelodiaANotasPrincipales1 :: DurMin -> Melodia -> [NotaPrincipal]
deMelodiaANotasPrincipales1 durMin melodia = reverse (drop 1 (deMelodiaANotasPrincipales1Rec
                                             durMin (C,0%1) (0%1) (reverse melodia) ))

deMelodiaANotasPrincipales1Rec :: DurMin -> NotaPrincipal -> Dur -> Melodia -> [NotaPrincipal]
deMelodiaANotasPrincipales1Rec durMin (pc, d) durAcumulada [] = [ (pc, d + durAcumulada) ]
deMelodiaANotasPrincipales1Rec durMin ultNotaP durAcumulada ( (Silencio d) : resto ) =
       deMelodiaANotasPrincipales1Rec durMin ultNotaP (durAcumulada + d) resto
deMelodiaANotasPrincipales1Rec durMin ultNotaP durAcumulada ( (Nota (pc, o) d) : resto )
       | d >= durMin  =  ultNotaP : deMelodiaANotasPrincipales1Rec durMin
                         (pc, d + durAcumulada) (0%1) resto
       | d <  durMin  =  deMelodiaANotasPrincipales1Rec durMin ultNotaP (d + durAcumulada)
                         resto
```

7.2.2. Un método más depurado

Este método es un poco más depurado que el anterior y selecciona a más notas principales. También tiene en cuenta cosas como el ritmo o hacia dónde se mueve una nota. Se basa en el criterio que expone Enric Herrera en su libro Teoría Musical y Armonía Moderna Volument 1 página 72. Paso a explicar los diferentes casos:

1. Nota de larga duración. Igual que en la sección anterior
2. Nota seguida de silencio de, al menos, su misma duración.
3. Nota seguida de otra entre las que existe un salto. Suponemos como salto la distancia en altura de más (estricto) de 2 semitonos.
4. Nota corta que resuelve en su inmediata inferior de tiempo fuerte a débil

Para saber si el tiempo en el que cae una nota es fuerte o débil hacemos lo siguiente. Necesitamos la duración de toda la canción anterior a esa nota, incluida la nota, y el denominador de la duración de dicha nota (a lo que hemos llamado *resolución*). Cambiamos la duración por una fracción equivalente que tenga de denominador la resolución. Si la nueva fracción tiene un numerador par entonces el tiempo es fuerte, si no es débil.

```
esTiempoFuerte :: Dur -> Resolucion -> Bool
esTiempoFuerte du r = even num
       where (num, den) = cambiaResolucion du r

cambiaResolucion :: Dur -> Resolucion -> (Int, Int)
cambiaResolucion du r = (newN , newD)
       where (n, d) = ( numerator du, denominator du);
             newN = n * (div r d);
             newD = r
```

Este procedimiento está pensado y probado para duraciones y resoluciones que sean potencia de dos, tal y como están preparadas las notas en música (es decir, 1/2, 1/4, 1/8, etc.). Es difícil conocer el acento del tiempo si no cumple las condicines ahora dichas y no me comprometo a que sea correcto el resultado en caso de que no las cumpla.

La función completa es la siguiente:

```
deMelodiaANotasPrincipales3 :: DurMin -> Melodia -> [NotaPrincipal]
deMelodiaANotasPrincipales3 durMin melodia = reverse (drop 1 (deMelodiaANotasPrincipales3Rec
       durMin (C,0%1) (0%1) (reverse (catalogaNotasPrincipales3 (0%1) durMin melodia)) ))

deMelodiaANotasPrincipales3Rec :: DurMin -> NotaPrincipal -> Dur -> [(HaskoreSimple, Bool)]
                                  -> [NotaPrincipal]
deMelodiaANotasPrincipales3Rec durMin (pc, d) durAcumulada [] = [ (pc, d + durAcumulada) ]
deMelodiaANotasPrincipales3Rec durMin ultNotaP durAcumulada ( (Silencio d, _) : resto ) =
        deMelodiaANotasPrincipales3Rec durMin ultNotaP (durAcumulada + d) resto
deMelodiaANotasPrincipales3Rec durMin ultNotaP durAcumulada ( (Nota (pc, o) d, b) : resto )
       | b == True  =  ultNotaP : deMelodiaANotasPrincipales3Rec durMin (pc, d + durAcumulada)
                                  (0%1) resto
       | b == False =  deMelodiaANotasPrincipales3Rec durMin ultNotaP (d + durAcumulada) resto

type DurAnt = Dur
catalogaNotasPrincipales3 :: DurAnt -> DurMin -> Melodia -> [(HaskoreSimple, Bool)]
catalogaNotasPrincipales3 _ _ [] = []
catalogaNotasPrincipales3 durAnt durMin (Nota (pc, o) d : resto )
       | d >= durMin = ( Nota (pc, o) d, True ) : catalogaNotasPrincipales3 (durAnt+d) durMin
                       resto
catalogaNotasPrincipales3 durAnt durMin (Nota (pc, o) d1 : Silencio d2 : resto)
       | d2 >= d1  = ( Nota (pc, o) d1, True ) : ( Silencio d2 , False) :
                     catalogaNotasPrincipales3 (durAnt+d1+d2) durMin resto
catalogaNotasPrincipales3 durAnt durMin (Nota (pc1, o1) d1 : Nota (pc2, o2) d2 : resto)
       | abs (pitchClass pc1 - pitchClass pc2) > 2 = ( Nota (pc1, o1) d1, True ) :
                     catalogaNotasPrincipales3 (durAnt+d1) durMin (Nota (pc2, o2) d2 : resto)
```

```
catalogaNotasPrincipales3 durAnt durMin (Nota (pc1, o1) d1 : Nota (pc2, o2) d2 : resto)
       | esTiempoFuerte (durAnt + d1) (denominator d1) && ((dist == 1) || (dist == 2)) =
                    ( Nota (pc1, o1) d1, True ) : catalogaNotasPrincipales3 (durAnt+d1)
                     durMin (Nota (pc2, o2) d2 : resto)
       where dist = (pitchClass pc1 - pitchClass pc2)
catalogaNotasPrincipales3 durAnt durMin (Nota (pc, o) d : resto ) =
        (Nota (pc, o) d , False) : catalogaNotasPrincipales3 (durAnt+d) durMin resto
catalogaNotasPrincipales3 durAnt durMin (Silencio d : resto) =
        (Silencio d, False) : catalogaNotasPrincipales3 (durAnt+d) durMin resto
```

La función *catalogaNotasPricipales* tiene la finalidad de asignar un booleano a cada nota indicando si es principal o no. Esto simplemente tiene la finalidad de hacer el algoritmo un poco más simple.

7.3. Armonización de notas principales

Una vez que ya tenemos las notas principales hay que buscar aquellos acordes que posean a esas notas principales. Cuando hablamos de poseer a las notas principales nos referimos exclusivamente a su *PitchClass*.

Hay que tener en cuenta una cosa importante. Suponemos que la melodía pasado está en la tonalidad de C Mayor, lo que significa que las notas C, D, E, F, G, A y B son diatónicas a la tonalidad y que el resto son notas cromáticas. Esto implica que las notas diatónicas van a ser armonizadas con acordes diatónicos mientras que el resto serán armonizadas con acordes cromáticos (dominantes secundarios, disminuidos, etc.).

Al igual que antes vamos a tener dos formas de armonizar:

7.3.1. Un acorde por nota principal

Es el caso más simple y musicalmente suena bien cuando las notas principales son relativamente largas.

Como el propio nombre indica a cada nota principal le va a asignar un unico acorde por lo que el procedimiento es bastante simple.

1. Para cada nota principal busca todos los acordes que posean a dicha noa (en funcion de si es diatonica o no).

2. De todos los acordes que la pueden armonizar se queda con uno aleatoriamente.

```
armonizaNotasPrincipales1 :: RandomGen g => g -> ModoAcordes
                             -> [NotaPrincipal] -> Progresion
armonizaNotasPrincipales1 _   _  [] = []
armonizaNotasPrincipales1 gen ma (notaP : resto) = cifradoYDur :
     armonizaNotasPrincipales1 sigGen ma resto
           where (cifradoYDur, sigGen) = armonizaNotaPrincipal1 gen ma notaP

armonizaNotaPrincipal1 :: RandomGen g => g -> ModoAcordes -> NotaPrincipal
                          -> ((Cifrado, Dur), g)
armonizaNotaPrincipal1 gen ma (notaP, dur) = ((cifradoAleatorio, dur), sigGen)
```

```
          where cifradosCandidatos = buscaCifradosCandidatosDeCMayor ma notaP
                (cifradoAleatorio , sigGen )= elementoAleatorio gen
                                              cifradosCandidatos
```

7.3.2. Más largo posible

La idea de este método es también bastante sencilla. Consiste en buscar el acorde que englobe el mayor numero de notas principales posible. Existe la posibilidad de introducir una duración máxima para ese acorde y así evitar que generen acordes demasiado largos aunque si buscamos un único acorde para armonizar toda una melodía (que puede no conseguirse) basta con poner esa duración muy grande. El algoritmo es como sigue:

1. Busca todos los acordes para armonizar una nota y continua con el resto de notas.
2. Para el resto vuelve a buscar los acordes candidatos y realiza la interseccion con la lista del punto 1. De esa forma nos vamos quedando con los acordes que armonizan las notas anteriores.
3. Continuamos asi hasta que dicha lista sea vacia o la duración del acorde exceda la duración maxima. En ese caso elegimos un acorde al azar (de la lista anterior porque la actual esta vacia, claro) y volvemos al punto 1.

```
type CifradosCandidatos = [Cifrado]
type DurAcordeAcumulado = Dur

-- Para comenzar con los cifrados condidatos
armonizaNotasPrincipales3 :: RandomGen g => g -> ModoAcordes -> DurMaxA
                             -> [NotaPrincipal] -> Progresion
armonizaNotasPrincipales3 _  _ durMaxA [] = []
armonizaNotasPrincipales3 gen ma durMaxA ( (pc, d) : resto)
     | d == durMaxA  =  (cifradoAleatorio, d ) : armonizaNotasPrincipales3
                        newGen ma durMaxA resto
     | d >  durMaxA  =  (cifradoAleatorio, durMaxA ) : armonizaNotasPrincipales3
                        newGen ma durMaxA ( (pc, d - durMaxA) : resto)
     | d <  durMaxA  =  armonizaNotasPrincipales3' gen ma durMaxA cifradosCandidatos
                        d resto
     where cifradosCandidatos = buscaCifradosCandidatosDeCMayor ma pc;
           (cifradoAleatorio, newGen) = elementoAleatorio gen cifradosCandidatos

-- Cuando ya tenemos una lista de cifrados candidatos
armonizaNotasPrincipales3' :: RandomGen g => g -> ModoAcordes -> DurMaxA ->
                              CifradosCandidatos -> DurAcordeAcumulado -> [NotaPrincipal]
                               -> Progresion
armonizaNotasPrincipales3' gen ma durMaxA cifCan durA [] = [(cifAlea, durA)]
    where (cifAlea, newGen) = elementoAleatorio gen cifCan
armonizaNotasPrincipales3' gen ma durMaxA cifCan durA ( (pc, d) : resto)
    | durA == durMaxA                        = (cifAlea , durMaxA) :
            armonizaNotasPrincipales3 newGen ma durMaxA ( (pc, d) : resto )
    | durA + d == durMaxA && newCifCan /= [] = ( newCifAlea , d + durA) :
            armonizaNotasPrincipales3 newGen2 ma durMaxA resto
    | durA + d == durMaxA && newCifCan == [] = ( cifAlea , durA) :
```

```
        armonizaNotasPrincipales3 newGen ma durMaxA ( (pc, d) : resto)
| durA + d <  durMaxA && newCifCan /= [] =
        armonizaNotasPrincipales3' gen ma durMaxA newCifCan (durA + d) resto
| durA + d <  durMaxA && newCifCan == [] = ( cifAlea , durA) :
        armonizaNotasPrincipales3 newGen ma durMaxA ( (pc, d) : resto)
| durA + d >  durMaxA && newCifCan /= [] = ( newCifAlea , durMaxA) :
        armonizaNotasPrincipales3 newGen2 ma durMaxA ( (pc, d - durMaxA + durA) : resto)
| durA + d >  durMaxA && newCifCan == [] = ( cifAlea , durA) :
        armonizaNotasPrincipales3 newGen ma durMaxA ( (pc, d) : resto)
where newCifCan = eliminaCifradosNoValidos cifCan (buscaCifradosCandidatosDeCMayor ma pc);
      (cifAlea, newGen) = elementoAleatorio gen cifCan;
      (newCifAlea, newGen2) = elementoAleatorio gen newCifCan;
```

7.4. Salida del módulo

La salida de este módulo es una progresión de acordes como ya hemos visto. En Genaro consiste en una lista formada por parejas de cifrado más duración. El cifrado indica la especie del acorde (ej, D m, Re menor) y la duración el tiempo que se extiende. Prolog también usa las mismas progresiones pero con otro formato ya que Prolog no entiende de tipos, solamente conoce términos (funciones y variables).

El interfaz gráfico está implementado en C++ y en un principio se pensó en que parseara progresiones en formato Prolog. Ahora que tenemos progresiones en formato Haskell podemos hacer dos cosas: que Haskell escriba la progresión haciendo un *show* y que se construya otro parser en el interfaz o usar el mismo parser pero escribir la progresión en formato Prolog. Se optó por hacerlo de esta última manera porque era la más fácil.

Por eso cuando se necesita que Haskell escriba en un fichero la progresión que ha generado el armonizador antes tiene que llamar a la función *deProgresionAString* y *escribeProgresionComoProlog* que se encuentran en el módulo *Progresiones.hs*.

Capítulo 8

Generador de líneas de bajo

8.1. Introducción

Este módulo se encarga de componer *líneas de bajo* a partir de progresiones de acordes. Una línea de bajo es una composición hecha para un bajo, no hay diferencia entre una línea de bajo y una melodía hecha para una parte de bajo, es solamente cuestión de terminología. Pero se emplea el término línea de bajo porque tiene ciertas connotaciones: de una línea de bajo se espera que sea rítmica y repetitiva, circular,que vaya más pegada al acorde, siguiendo al acorde de forma más subordinada que una melodía; y que se coordine bien con la batería, que se *empaste* bien con ésta. Es decir, se espera que desempeñe funciones rítmicas y armónicas. De una melodía en cambio se espera más linealidad, mayor libertad rítmica y mayor independencia de las notas estables del acorde.
Al igual que con las melodías, se emplean las reglas de la armonía para que la línea de bajo resultado este relacionada con la progresión y 'quede bien' al sonar a la vez que la traducción a música de ésta. Y también como las melodías, las líneas de bajo producidas por Genaro son monofónicas.
Hay 3 bajistas correspondientes a 3 algoritmos de generación de bajo:

1. El bajista *Fundamentalista* es casi determinista, y acompaña a cada acorde tocando su nota fundamental (la ms estable del acorde). Solamente hay aleatoriedad en la elección de la octava empleada para cada acorde.

2. El bajista *Aphex* compone aplicando mutaciones aleatorias similares a las de la melodía, sobre un bajo compuesto por el fundamentalista.

3. El bajista *Walking* interpola las notas del bajista fundamentalista haciendo que las notas intermedias tengan una duración especificada, y luego muta el resultado de forma similar a Aphex.

Cada bajista se diseñó extiendo al anterior, aumentando el nivel de complejidad del algorítmo.

8.2. Abstracciones empleadas

Este módulo fué de los últimos en hacerse y por ello se beneficia de las lecciones aprendidas durante el desarrollo de los demás módulos. En cambio, por ser uno de los últimos módulos es que se hizo con más prisa y quizás por ello no se desarrollaron abstracciones adicionales para éste módulo. O quizás es

que tampoco son muy necesarias, visto que las mutaciones aplicadas a listas de músicas funcionan bien para simular variaciones de una melodía. Es un tema sobre el que habría que pensar con calma en las ampliaciones futuras de genaro, en todo caso la versión actual del generador de líneas de bajo de Genaro utiliza:

1. Curvas melódicas: solamente el bajista Walking las emplea, generándolas de una manera muy concreta que se explicará en la sección de implementación.

2. Progresiones de acordes: todos los bajistas hacen lo mismo en realidad, tocar las fundamentales de los acordes y meter más o menos notas entre medias.

3. No utiliza listas de acentos: El ritmo se produce de distintas maneras según el bajista:

   a) Fundamentalista: lo marca la progresión y nada más

   b) Aphex: lo marca la progresión y las mutaciones, que dividen el tiempo en potencias de dos, algo apropiado en el contexto binario en que trabaja Genaro. La aplicación de un número suficiente de mutaciones produce variaciones rímticas muy ricas.

   c) Walking: determinado fundamentalmente por la duración que se le pasa de parámetro, las mutaciones pueden también añadir variaciones rítmicas interesantes.

Este módulo, como el de la melodía, también utiliza el concepto de *registro*, que recordemos, es una restricción sobre las alturas de las notas, que deben pertenecer a un intervalo determinado por éste. El bajo tiene asociado un registro grave, como es lógico.

8.3. Algoritmo de generación de líneas de bajo

Como en el caso de la generación de melodías, lo difícil es generar la música para un sólo acorde, después generar la música para toda la progresión consiste simplemente en enlazar la música para cada acorde. La idea fundamental en la que se basan los algorítmos es que, recordemos, *todos los bajistas hacen lo mismo en realidad, tocar las fundamentales de los acordes y meter más o menos notas entre medias*. Teniendo eso en cuenta, veamos por separado los algoritmos para cada tipo de bajista:

1. Aphex: Este algoritmo recibe como entrada dos alturas correspondientes a la fundamental del acorde para el que hace la línea, y al acorde siguiente. Por tanto son las alturas entre las que le bajista debe insertar notas. También recibe el tipo de escala correspondiente al acorde, y la duración de éste, a la que se deberá ajustar la línea. Por último recibe una serie de parámetros enteros. El algortimo funciona como sigue:

   (i) Primero construye una lista de dos notas compuesta por una nota de altura igual a la primera suministrada, y de duracón igual a la suministrada, seguida de otra nota de altura igual a la segunda suministrada y de una duración cualquiera (se pone siempre la misma pero no importa). Las listas de notas son la entrada a los algoritmos mutadores, así que con esta lista, que es una primera versión de la línea de bajo, ya tenemos algo que mutar.

   (ii) Se muta esta primera línea de bajo con las mutaciones que introducen notas intermedias, que son las mismas mutaciones *dividir notas* y *dividir notas fino* que empleábamos para la melodía. El número de veces que se aplican y el parámetro de la segunda mutación vienen dados por los enteros entrada del algoritmo. Y se aplican de forma entrelazada (no se aplica

primero n_1 veces una mutación y luego n_2 veces la otra, sino que cada vez se elige aleatoriamente cuál de las dos aplicar, dando la misma probabilidad a las dos, hasta que se hayan aplicado el número de veces especificado por los parámetros.

(iii) Se elimina la última nota de la línea mutada. Esto se hace porque esa última nota era la correspondiente a la fundamental del acorde siguiente, y en realidad no pertenece a esta línea sino a la del acorde siguiente. Además la duración de esa nota se puso sin seguir ningún criterio, porque se sabía desde un principio que iba a terminar eliminándose. Sin embargo era necesario poner esa nota porque las mutaciones de notas intermedias necesitan tener dos alturas entre las que poner notas.

(iv) Se muta esta nueva línea con una variación de la mutación 'alargar notas' de la melodía llamada *alargar notas destructivo*. Esta mutación es similar, pero considera candidato a alargarse cualquier nota no silencio que sea seguida por cualquier nota, sea silencio o no. Después elige uno solo de los candidatos, dando probabilidad mayor de ser elegido a los que corresponden a alturas más estables dentro del acorde. Despúes alarga la nota elegida eleminando la nota que la sigue y sumando la duración de ésta a la de la nota elegida. Se aplica tantas veces como indica el parámetro de entrada correspondiente.

2. Fundamentalista: El bajo fundamentalista en realidad no tiene un algoritmo propio, sino que es un caso derivado del bajista walking en el que el número de veces que se aplican las mutaciones es 0.

3. Walking: Este algoritmo recibe como entrada la misma información que el bajista Aphex, añadiendo también la duración de las notas de la línea, y el desplazamiento máximo que se empleará para generar la curva. Estos dos parámetros se explicarán mejor cuando aparezcan en el algoritmo, que funciona como sigue:

   (i) Se calcula la distancia en la escala entre las dos alturas de entrada, entre las que se quiere generar la línea. Con esta distancia se construye la curva melódica que tiene esta distancia como único punto. Esta curva representa el salto desde la primera altura a la segunda.

   (ii) El parámetro que indica la duración de las notas de la línea indica la duración que tendrán casi todas las notas de la línea, antes de mutar la línea con la mutaciones usadas en el bajista Aphex. Decimos casi todas porque es posible que la duración a la que hay que ajustar la línea no sea divisible por la duración de las notas y haya que añadir una sola nota de otra duración para ajustarla. Teniendo esto en cuenta se calcula cuantas notas de esa duración (más quizás una adicional) necesitamos para ocupar la duración del acorde.

   (iii) Aplicamos a la curva melódica inicial una mutación especial que desplaza un punto de la curva elegido al azar, una cantidad aleatoría entre -(desplazamiento máximo) y +(desplazamiento máximo); y que después inserta en un posición de la curva elegida al azar otro punto de valor contrario al desplazamiento que se realizó. De esta manera se consigue otra curva con un punto más y que realiza un desplazamiento total (suma de todos los puntos de la curva) igual al de la curva de partida. Aplicamos esta mutación a la curva inicial un número de veces igual al número de notas calculadas en el paso (ii) menos uno, para obtener una curva melódica con número de puntos igual a ese número de notas.

   (iv) A esta curva melódica se le quita el último elemento y se le añade de primer elemento un 0, para que al aplicarla sobre la altura del acorde la primera altura que aparezca sea dicha altura.

(v) Construimos una lista de duraciones para las notas de la línea. Será una lista cuyo único elemento sea la duración del acorde si éste dura menos que la duración especificada para las notas. Si no, si la duración del acorde es divisible por la de las notas será una lista en la que todos los elementos son la duración de las notas. Si no es divisible será una lista en la que todos los elementos menos uno son la duración de las notas, y en la que el elemento distinguido será la duración que se añade para encajar la línea a la duración del acorde. La posición de ese elemento distinguido se elegiría al azar.

(vi) Se mezclan las dos listas generadas en los puntos (iv) y (v) para conseguir una primera versión de la línea de bajo.

(vii) Esta primera versión de la línea de bajo se muta tantas veces como indiquen los parámetros, aplicando las mutaciones siempre en este orden: alargar notas destructivo, dividir notas y dividir notas fino.

Una vez sabemos generar una línea de bajo para una acorde generarla para una progresión es sencilla. Se distribuyen las mutaciones entre los acordes según sus duraciones y se llama al algoritmo de generación de líneas para acordes correspondiente al bajista especificado. Las líneas siempre van de un acorde al siguiente excepto para el último acorde que enlaza con el primero.

8.4. Implementación

Se apoya en la librería Haskore de Haskell, y está desarrollada íntegramente en este lenguaje. También se apoya mucho en el funciones desarrolladas para el módulo de melodías.

El único tipo nuevo definido fué el tipo de bajista:

```
data TipoBajista = Aphex | Walking | Fundamentalista
     deriving(Enum,Read,Show,Eq,Ord,Bounded)
```

8.5. Otros usos de éste módulo

Al escuchar las músicas resultado resulta sorprendente observar la capacidad melódica del bajista Aphex, que en ocasiones es mejor melodista que el módulo de melodía. En un futuro quizás podría ofrecerse como otra opción de generación de melodías.
La combinación de varios bajistas Walking a distintas velocidades (es decir, para distintas duraciones de nota) da como resultado músicas que recuerdan a las polifonías barrocas. Podría ser un punto de partida para empezar un módulo con esta dirección.

8.6. Capacidad de ampliación

Como siempre, la ausencia de objetos e interfaces y las abstracciones definidas favorecen la ampliabilidad de este módulo, proporcionando puntos de acceso para la generación de melodías. Las vias más evidentes por las que se podría ampliar este módulo son:

1. Adición del registro como parámetro de la generación de líneas, para poder generar líneas más agudas y emplearlas como melodías.

2. Uso de cromatismos en el bajo: utilizar alturas muy inestables, que no pertenecen a la escala, como *notas de paso* en las líneas. Esto dotaría de más tensión y moviento a la música.

3. Algo totalmente diferente: Cualquier programa en cualquier lenguaje que lea archivos de texto en el formato especificado por el predicado es_progresion/1, y ficheros de patrón rítmico puede sustituir a este módulo.

Capítulo 9

Batería

9.1. Introducción

La percusión es la parte instrumental que va a marcar el ritmo. Bebe de las ideas de los *patrones rítmicos* y de los *acordes ordenados*, por tanto, si esas partes se han entendido no va a costar ningún trabajo entender ésta.

9.2. Los instrumentos de la batería de Genaro

Los intrumentos que va a tener Genaro en su batería son los siguientes:

```
data PercusionGenaro = Bombo | CajaFuerte | CajaSuave | CharlesPisado | CharlesAbierto
                     | CharlesCerrado | TimbalAgudo | TimbalGrave | Ride | Crash
    deriving (Show, Read, Eq, Ord, Enum, Ix)
```

que son guardados en la lista

```
bateriaGenaro :: [PercusionGenaro]
bateriaGenaro = [ toEnum i | i <- [0..9] ]
```

9.3. Implementación

9.3.1. Percusión en Haskore

El Haskore de Hudak posee su propia definición de tipos para la percusión que se encuentra en el tipo *PercussionSound*. Dicha asociación es una biyección con los intrumentos de percusión que posee MIDI. En el estandar MIDI cada instrumento de percusión esta asociado a una altura (*Pitch*, es decir, tipo de nota y octava) que es tocado por el canal 10.

Nosotros vamos a usar parte de esos instrumentos pero le vamos a dar otro nombre por comodidad. Por consiguiente tenemos que hacer una función de asocie bidireccionalmente ambas nomenglaturas.

```
bateriaHaskore :: [PercussionSound]
bateriaHaskore = map de_PercusionGenaro_a_PercusionHaskore bateriaGenaro
```

```
asociacionPercusion :: [(PercusionGenaro, PercussionSound)]
asociacionPercusion = [(Bombo, BassDrum1),
                       (CajaFuerte, LowMidTom ),
                       (CajaSuave, HiMidTom ),
                       (CharlesPisado, PedalHiHat ),
                       (CharlesAbierto, OpenHiHat ),
                       (CharlesCerrado, ClosedHiHat ),
                       (TimbalAgudo, LowTimbale ),
                       (TimbalGrave, HighTimbale ),
                       (Ride, RideCymbal1 ),
                       (Crash, CrashCymbal1 )]



de_PercusionGenaro_a_PercusionHaskore :: PercusionGenaro -> PercussionSound
de_PercusionGenaro_a_PercusionHaskore pg = ph
      where (lpg, lph) = unzip asociacionPercusion;
            Just ind = elemIndex pg lpg;
            ph = lph !! ind


de_PercusionHaskore_a_PercusionHaskore :: PercussionSound -> PercusionGenaro
de_PercusionHaskore_a_PercusionHaskore ph
      | isJust quizas_ind    = lpg !! (fromJust quizas_ind)
      | isNothing quizas_ind = error ("de_persucionHaskore_a_PercusionHaskore "
            ++ show ph ++ ":No se puede traducir a PercusionGenaro")
      where (lpg, lph) = unzip asociacionPercusion;
            quizas_ind = elemIndex ph lph
```

9.3.2. Usando conocimientos anteriores

Vamos a usar aquello que tenemos hecho de patrones rítmicos y de acordes ordenas para formar la batería.

Como ya hemos dicho antes cada instrumento de percusión de MIDI es representado por una altura (pitch). La unión de todos ellos en una lista junto con una duración obtendriamos un acorde ordenado:

```
acordeOrdenadoBateria :: Dur -> AcordeOrdenado
acordeOrdenadoBateria d = (map f bateriaHaskore,d)
       where f = pitch.(+35).fromEnum

acordeOrdenadoBateria' :: Dur -> [AcordeOrdenado]
acordeOrdenadoBateria' d = [acordeOrdenadoBateria d]
```

Si ahora le aplicamos a ese acorde ordenado un patrón rítmico vamos a repartir dichas notas en el tiempo.

```
encajaBateria :: Dur -> PatronRitmico -> Music
encajaBateria d pr = deAcordesOrdenadosAMusica Ciclico (Truncar1, Truncar2)
                     pr (acordeOrdenadoBateria' d)
```

El patrón rítmico pasado, aunque puede ser cualquiera, debe ser uno que tenga 10 voces de altura, es decir, tantas como instrumentos de percusión. Es por ello que los patrones de encaje verticales nos son indiferentes ya que encaja perfectamente.

Se podría haber hecho que aceptara cualquier patrón pero las diferentes formas de encaje no tienen aqui tanto sentido como cuando el acorde ordenado representa notas de un instrumento. Por tanto se descarto esa posibilidad, no aqui, en Haskell, sino en el interfaz gráfico.

Por ultimo sólo queda que el *Music* de salida sea ejecutado por el instrumento "Drums" de Haskore para que sea enviado al canal 10 cuando se pase a MIDI. Pero esta parte no se hace en el módulo *Bateria.hs* sino en el *main.hs* que es el módulo con el que se comunica el interfaz.

Veamos un ejemplo:

```
bateriaMusic :: Music
bateriaMusic = Instr "Drums" (encajaBateria (2%1) patronRitmicoCualquiera)
```

9.4. Resultado final

Al final Genaro no tiene batería por falta de tiempo. Quizás un tiempo de dedicación de una hora o dos hubiera bastado para tenerla termina.

9.5. Posibles mejoras

Como habrá observado el avispado lector la batería de Genaro no tiene nada de aleatoriedad. Hubiera sido interesante meter algo de ello. Las ideas que se me ocurrieron a mí con ayuda de compañeros de clase son las siguientes:

1. Que las partes fuertes de un compás sean tocadas por un instrumento o varios (por ejemplo el bombo) y las débiles por otro. Una vez hecho esto se podría mutar ligeramente para que se note continuidad a la vez que cambio.
   Una mutación que se me ocurre es cambiar un instrumento por otro. Por ejemplo, si el bombo toca la parte fuerte hace que pase a tocarla el crash.

2. Hacer que la batería se adapte a la duración de un acorde. Ya que los acordes no tienen la misma duración podríamos hacer, por ejemplo, que el primer tiempo fuerte sea tocado por el bombo y el resto por una caja.

3. Hacer que la batería se adapte a una progresión de acordes. Una técnica de grupos modernos es intercambiar los intrumentos que tocan en la parte fuerte y débil en los últimos compases de la progresión. Por ejemplo, si una caja se golpea en tiempo fuerte que pase golpearse en tiempo débil. Ello da sensación de que algo va a cambiar.

Capítulo 10

Haskore a Lilypond

10.1. Introducción

Este módulo pretende ser un traductor que pasa del árbol de constructoras que representa el tipo *Music* a un *String* que sea entendido por Lilypond. Nosotros hemos usado la version 2.4.3 de Liypond para compilar.

En primer lugar ibamos a usar el programa *midi2ly* que viene junto con Lilypond y que pasa un archivo MIDI a formato Lilypond. Las pruebas que hicimos con ellos nos desilusionaron bastante ya que el resultado obtenido estaba lejos de ser correcto incluso con ejemplos simples (hechos con Haskore, es decir, que no estaba la mano de ningún humano que podía variar el tiempo en milésimas).

Es por ello que intentamos hacer la traducción nosotros mismos a algun formato que se pudiera compilar a pdf. A diferencia de otros lenguajes como musicTex o MusiXTEX, el formato de Lilypond era muy parecido a la estructura del tipo Music de Haskore. Esto nos motivó a hacer la conversión de esta forma.

10.2. Traducción

En esta sección no vamos a entrar en cómo es el proceso de conversión completo ni en todas las características de Lilypond porque, más que nada, ni yo mismo las conozco.

Ya que el tipo Music no posee información sobre la obra completa hemos tenido que hacer un tipo que lo englobe. Ese tipo se llama *CancionLy* y es muy simple de entender:

```
type Armadura = (PitchClass, Modo)
data Modo = Mayor | Menor
     deriving (Eq, Ord, Show, Read, Enum)
type Ritmo = (Int  -- Numero de notas del compas
             ,Int  -- Resolucion del compas
             )
type Instrumento = String
data Clave = Sol   -- Clave de 'Sol'
             | Fa  -- Clave de 'Fa'
             | Bateria -- Clave de 'Bateria'
```

```
type Score = (Music, Armadura, Ritmo, Instrumento, Clave)
type Titulo = String
type Compositor = String

type CancionLy = (Titulo, Compositor, [Score])
```

Muchas de las cosas anteriores se comprenden por lo que no vamos a entrar a detallarlas. La parte más importante es la traducción de CancionLy es la parte del Score (pentagrama en inglés) que contiene al tipo Music.

10.2.1. Traducción del tipo Music

Como hemos dicho, Lilypond y Music tienen varias cosas en común que hacen atractiva su conversión. En concreto son dos: ambos poseen un operador para secuenciar música y para paralelizarla.

El operador de música paralela (que se interpreta a la vez) es el :=: en Haskore. En Lilypond es todo aquello que va entre << y >>. El operador secuencia en Haskore es el :+: y en Lilypond es todo aquello que no está entre << y >>. Lilypond también es capaz de agrupar música poniendo llaves { }, a lo que llama secuencia.

```
deMusicALy :: Music -> String
deMusicALy (Note p dur _) = imprimeNota p dur
deMusicALy (Rest dur)     = imprimeSilencio dur
--   Cambia el operador secuencia de Haskore por el secuencia de Lilypond
deMusicALy (m1 :+: m2)    = " { " ++ deMusicALy m1 ++ " " ++ deMusicALy m2 ++ " } "
--   Cambia el operador paralelo de Haskore por el paralelo de Lilypond
deMusicALy (m1 :=: m2)    = " << " ++ deMusicALy m1 ++ " " ++ deMusicALy m2 ++ " >> "
deMusicALy (Tempo d m)    = "\\times " ++ imprimeRatio d ++ " { " ++ deMusicALy m ++ " } "
--   Elimina la constructora 'Trans'
deMusicALy (Trans t m)    = deMusicALy (suma t m)
--   El resto de las constructoras son ignoradas
deMusicALy (Instr _ m)    = deMusicALy m
deMusicALy (Player _ m)   = deMusicALy m
deMusicALy (Phrase _ m)   = deMusicALy m
```

Esta es la función central del módulo *HaskoreALilypond.hs*. Algunas constructoras de Music tiene que ser ignoradas porque o no corresponden con nada de Lilypond o la traducción sería tan complicada que llevaría mucho tiempo y no merecería la pena.

Traducción de alteraciones

Cuando hablamos de alteraciones nos referimos a los sostenidos y bemoles musicales. Como el sistema del tipo Music es atemperado no importa si se ejecuta # C o bC ya que son el mismo sonido. Un músico de verdad no usaría ambos símbolos indistintamente porque no tienen el mismo significado. El problema es que no sabemos cómo se usa un símbolo u otro por lo que optamos por una solución simple y elegante: si la armadura tiene sostenidos entonces que sean todas las alteraciones sostenidos, e *idem* para bemoles.

La partitura final puede que no sea correcta musicalmente hablando pero, al menos, es más fácil de leer. La función que lo lleva a cabo es *arreglaAlteraciones* que no copio aquí por ser demasiado extensa.

Traducción de duraciones

Por último hay que señalar un apunte sobre la traducción de duraciones de notas o silencios. Lilypond sólo entiende duraciones que son potencias de dos, es decir, 1 es la redonda, 2 la blanca, etc. Pero las duraciones en Haskore son fracciones de enteros. Por ello se ha hecho lo siguiente:

1. Si el numerador es mayor que uno entonces se repite tantas veces como diga dicho numerador y todas las notas se ligan.
2. Si el denominador no es potencia de 2 se cambia a la potencia de dos más cercana. Este último paso elimina informacion del tipo Music pero no se qué podríamos hacer si llega algo del estilo de 1/27. En caso de dejarselo a lilypond este siempre pone una redonda cuando no lo entiende, cosa que es todavía peor.

Esto ha motivado la siguiente funcion:

```
imprimeNota :: Pitch -> Dur -> String
imprimeNota p dur
   -- Ponemos puntillo
   | numerador == 3 && denominador > 1 = imprimePitch p ++
                                         show (quot denominador 2) ++ "."
   | otherwise                         = eliminaUltimos 2 (concat [imprimePitch p ++
                                         show (redondeaAPotenciaDos(denominator dur)) ++
                                         "~ " | i <- [1..numerador] ] )
   where numerador = numerator dur;
         denominador = denominator dur;
```

10.3. Problemas

El principal problema no ha sido la conversión al formato Lilypond. Sabemos que muchas veces, sobre todo en melodía y bajo, existe mucha probabilidad de que el denominador de las notas no sea potencia de dos. Nos hubieramos conformado con que tuvieramos una ligera aproximación a las notas que genera Genaro.

El principal problema es el propio Lilypond. La version 2.4.2 estaba mal implementada y no conseguía compilar nada. La version 2.4.3 compilaba bien pero sólo cosas pequeñas. Cuando la canción era grande o el Music tenía forma de árbol degenerado (es decir, de una lista) que era lo que sucedia cuando pasabamos un MIDI a Music con las funciones que proporciona Haskore (*loadMidiFile* y *readMidi*) entonces daba un error de pila. Suponemos que es porque la conversión generaba muchas llaves { } que producían muchas llamadas recursivas en Lilypond.

10.4. Posibles ampliaciones

Una ampliación muy interesante habría sido introducir los cifrados que genera Genaro en la propia partitura y no sólo las notas. Lilypond puede hacerlo aunque habría que hacer la conversión e introducir más información en CancionLy.

Ya menos interesante, aunque posible, habría sido introducir característica de articulación, como puede ser el *legato*, o cosas como el *tempo* o la intensidad de la nota (seria traducir el *velocity* a *forte*, *mezoforte*, etc.)

Capítulo 11

Herramientas auxiliares

11.1. Timidity++

Timidity es un sintetizador software, realizado por Tuukka Toivonen, Masanao Izumo. Lo hemos usado por ser GNU.

11.2. swi prolog

Interprete y compilador de prolog, también es GNU. Su opción de compilar archivos prolog a ejecutables ha sido de gran ayuda para el intercambio de información.

11.3. Lilypond

Es un compilador de un lenguaje de descripción de partituras a .pdf o formato similar. También es GNU.

11.4. C++ builder

Es un compilador de C++ que posee facilidades para generar una GUI.

11.5. Hugs 98

Es un interprete de Haskell. Tiene una licencia que permite la libre distribución.

11.6. Haskore

Haskore son unas librerías de haskell que brindan facilidades para el tratamiento musical. Se fundamenta en el tipo *Music*, que es un tipo recursivo que representa la música. También proporciona

funciones para generar archivos midi a partir de elementos de tipo Music, y viceversa. El tipo Music está definido de la siguiente manera:

```
data Music = Note Pitch Dur [NoteAttribute]   -- a note \ atomic
           | Rest Dur                         -- a rest /    objects
           | Music :+: Music                  -- sequential composition
           | Music :=: Music                  -- parallel composition
           | Tempo  (Ratio Int) Music         -- scale the tempo
           | Trans  Int Music                 -- transposition
           | Instr  IName Music               -- instrument label
           | Player PName Music               -- player label
           | Phrase [PhraseAttribute] Music   -- phrase attributes
    deriving (Show, Eq)
type Dur   = Ratio Int                        -- in whole notes
type IName = String
type PName = String
```

Capítulo 12

Estado del arte en la composición aleatoria de música

12.1. Introducción

En esta sección haremos un pequeño recorrido por los sistemas que hay para la creación de música, y la manera en la que trabajan.

12.2. Programas que juegan con la música

Existen algunos programas que permiten crear o modificar piezas. Hablaremos de algunos de ellos para conocer un poco el campo de la creación de música antes de llegar nosotros.

12.2.1. Juego de los dados de Mozart

Uno de los primeros sistemas de composición automática que conocemos, fue el juego que ideó el músico *Wolfgang Amadeus Mozart*. Con este juego puedes componer cientos de minuetos, seleccionando distintos fragmentos musicales de unas tablas mediante el resultado de una pareja de dados.

12.2.2. JAMMER

JAMMER es un programa que permite generar, mediante parámetros especificados, acompañamientos musicales o arreglos a una canción. JAMMER no crea una canción de la nada, trabaja con una pieza que altera para obtener una nueva. Puedes conocer más sobre JAMMER en http://www.soundtrek.com.

12.2.3. KeyKit

KeyKit es un lenguaje de programación musical creado por Tim Thompson. Es la base para algunos programas de composición de música que enumeraremos a continuación:

Muse-O-Matic	El usuario ha de introducir una palabra, y obtiene como resultado una canción que se ha creado basandose en esa palabra. Este algoritmo es determinista, para una palabra generará siempre la misma canción.
Web Tones	Genera una pieza musical a partir de una página web.
Key Chain	Permite elegir al usuario entre distintas transformaciones que aplica a la canción.
Pieces-O-MIDI	Toma un midi y lo descompone en piezas, que mezcla para generar un nuevo midi
GIF Jam	Toma una imagen en formato gif, y crea un fichero midi dependiendo de los colores de cada pixel
Expresso	Este programa usa algoritmos fractales para conseguir funciones bastantes complejas en términos de una variable X, una nota sustituye la variable X y se evalúa la expersión.
Fresh Roast	Es un refinamiento de *Expresso*, que incluye baterías.
Life Forms	Se basa en el *Juego de la vida* para crear la canción.

Capítulo 13

Gestión del proyecto

13.1. Mantenimiento del código

Debido a la cantidad de código que se iba a generar, y a los problemas de integración que pudieran surgir, se decidió usar un sistema de mantenimiento de código llamado *CVS*. el CVS nos permitía trabajar desde nuestras casas y mantemer actualizadas las últimas versiones de los ficheros fuente. Debido a que el CVS nos permite almacenar no sólo código, lo hemos empleado también para guardar cualquier otro dato de interés para el proyecto, tales como documentos, actas de reuniones, ejemplos...

La estructura de directorios en el CVS es la siguiente:

Codigo	Aquí se encuentran los ficheros fuente de los distintos lenguajes.
Documentos	Se usa para guardar cualquier tipo de documento de interés, tales como actas, manuales...
Ejemplos	Donde guardamos los ejemplos que consideramos mas representativos del momento.
PatronesRitmicos	Donde se encuentran los patrones rítmicos creados para ser usados por GENARO.
PatronesBateria	Donde se encuentran los patrones de batería que iba a usar GENARo.
ProgramasAux	En el que se han guardado otros programas que se usan con GENARO
Timidity	Con el timidity configurado para ser invocado por GENARO.

El servidor CVS nos ha sido proporcionado por Berlios, http: developer.berlios.de

13.2. Reuniones

Con el fin de poner en común los avances semanales, así como de hacer una pequeña evaluación de la situación en cada momento de GENARO, se fijaron reuniones semanales que venían definidas en la *Gestión de la configuración*, documento que se encuentra en su correspondiente directorio del CVS. De cada una de estas reuniones hay un acta, con el nombre de la fecha en la que tuvo lugar la reunión. Este

acta era escrita y pasada a limpio por un secretario, puesto que se cambiaba semanalmente entre los miembros del equipo.

También se realizaron diversas reuniones con el director del proyecto, se intentaban realizar cada 2 semanas, pero en algunas ocasiones, debido a que los avances no eran los esperados, estas reuniones se retrasaban.

13.3. Comunicación interna

Para mantener el contacto entre los distintos miembros del grupo, se creo una lista de correo electrónico para poder difundir con la mayor brevedad posible una noticia a todos los integrantes. Esta lista nos la ofrece Berlios, tras registrar allíñuestro proyecto.

En ocasiones extraordinarias la comunicación se realizaba mediante llamadas telefónicas, a los numeros que vienen en la *Gestión de la configuración*.

13.4. Organización de trabajo

Se ha intentado asignar el trabajo individual de la manera más eficiente posible, para que cada miembro se distribuya su tiempo como mejor considere. Esta asignación se realizaba en las reuniones semanales. En ciertas ocasiones, el trabajo tenía que ser realizado de manera colectiva, en casos como enlazar distintas partes del proyecto. Para ello nos reuníamos en un mismo local todos los integrantes para codificar el código necesario.

Capítulo 14

Revisión de objetivos

14.1. Evaluación de objetivos

La finalidad de esta sección es revisar los objetivos que en su día fueron propuestos para GENARO, y revisar cuales de ellos han sido cumplidos, cuales no, y cuales han sido alcanzados sólo en parte.

14.1.1. Objetivos principales

Componer una canción para un tempo, tonalidad, escala y duración aproximada

GENARO es capaz de usar estos parámetros para generar una canción, a excepción de la tonalidad, para la que nos hemos restringido a tonalidades mayores. Esto es lo que queríamos obtener con GENARO.

Posibilidad de reproducir la música generada a través de nuestro programa

Con la ayuda de un programa auxiliar llamado *Timidity*, que es invocado de manera transparente al usuario, somos capaces de reproducir la música que genarmos a través de nuestro programa.

Exportar la música generada a formatos .wav y .midi

A través de las librerias de *Haskore* GENARO crea un midi con la música que ha compuesto. Este midi puede ser exportado a un fichero .wav mediante la ayuda del *Timidity*, que es invocado nuevamente por el programa principal.

Interfaz gráfica amigable

El interfaz gráfico facilita de manera intuitiva un control absoluto sobre los parámetros de uso de GENARO a un usuario. Actúa como nexo ante los demás programas para facilitar al usuario el uso de GENARO.

14.1.2. Objetivos secundarios

Generar un archivo pdf o ps con la partitura del instrumento elegido

Objetivo no cumplido. Por problemas de tiempo, de dificultad en el uso de Lilypond y en la traducción del tipo *Music* de *Haskore* no se ha podido introducir adecuadamente en Genaro. Hay algo hecho pero no es definitivo.

Edición de la estructura de la canción especificando para cada sección musical su tipo de escala y tonalidad, su tempo y otros atributos

Como hemos dicho los parametros de escala, tempo y tonalidad estan establecidos para la canción completa sin que se puedan cambiar en secciones individuales. Sin embargo cada seccion musical tiene propios parametros en funcion del tipo de pista que sí que se guardan y se editan separadamente.

Poder seleccionar qué instrumentos se silencian y cuáles no para poder oír y exportar en *.wav* cada parte por separado

Objetivo cumplido. Tanto cada sección musical como cada pista se pueden silenciar para que no se tengan en cuenta en la exportación a .wav.

Incorporar un editor de patrones rítmicos y otro de secuencias de acordes, y la posibilidad de salvarlos en ficheros. Posibilidad de elegir qué patrones y secuencias se van a considerar en la generación de la música

Objetivo cumplido. Se desarrollo un editor de patrones rítmicos independiente del interfaz principal, a diferencia del editor de secuencias de acordes (llamado progresión) que sí está ligado al interfaz principal.

Posibilidad de introducir una melodía por medio de la interfaz para que nuestro generador la armonice y componga un rítmo para ella

No está implementado del todo. Existe la posibilidad de armonizar una melodía generada por GENARO pero no la posibilidad de introducir una melodía por el interfaz. No sería muy difícil añadirlo ya que tenemos formularios para crear editores de pianola.

Sistema de comandos: se podrá invocar al programa a través de la consola con distintos parámetros, ofreciendo acceso a las mismas funcionalidades que la interfaz gráfica. El objetivo de esto es facilitar la reutilización del software

Definitivamente no esta implementado ni se pretendió implementarlo. Se le dio menos prioridad que el resto de las tareas ya que se refería a versatilidad de uso y no a las propias funciones de GENARO. Por falta de tiempo y debido a que su implementación sería bastante complicada no se hizo.

14.2. Ampliaciones posibles

Considerando el estado actual del proyecto, GENARO podría continuar avanzando en varios puntos. Aquí intentaremos enumerarlas, pero es muy recomendable acudir a los capítulos correspondientes a cada módulo para obtener una descripción más precisa y concreta de sus posibles ampliaciones.

14.2.1. Batería

Enlazar la batería a GENARO, e introduciendole aleatoriedad.

14.2.2. Patrones Rítmicos

Introducir patrones rítmicos dinámicos y aleatorios.

14.2.3. Acordes

Introducción de otras tonalidades y ritmos

14.2.4. Editor de pianola

Permitir ligar las notas de una misma voz, y crear un editor de pianola específico para la melodía.

14.2.5. Interfaz principal

Pulir el interfaz para que goce de mayor intuitividad, y añadirle las opciones de borrar pistas y bloques.

14.2.6. Compositores para otras texturas

A parte de la ya implementada melodía acompañada, compositores para otros estilos como fugas, canon, y en general estructuras con reglas muy rígidas.

14.3. Evolución del proyecto

Al inicio del proyecto no se tenían muchas ideas claras como es lógico. Pero si que había un par de intenciones bastante decididas:

1. Conseguir un compositor automático: la idea era que Genaro fuera un programa que apretando un botón compusiera una obra completa y coherente. No se habían fijado los instrumentos pero las ideas del trio de Jazz y el reparto de funciones rítmicas, melódicas y armónicas ya se habían tomado como punto de partida.

2. Emplear Sicstus Prolog para los algortimos de generación de música: el primer problema de generación de música que se abordó fue el de la generación de progresiones de acordes, porque la música en Genaro siempre se abordó desde la perspectiva de la melodía acompañada, y de la melodía compuesta como una improvisación sobre una base de acompañamiento, otra vez basándose en conceptos del Jazz.
   La generación de progresiones es el primer paso para la generación de acompañamientos porque está a un grado de abstracción más elevado que la música concreta, y permite definir el entorno musical o contexto armónico que caracteriza un fragmento musical, con lo que se pueden utilizar las progresiones para componer a partir de ellas otros fragmentos musicales que se reproduzcan a la vez que el acompañamiento, como el bajo y la melodía. Además tiene la ventaja de estar bastante

bien formalizada por las reglas de la armonía lo que hace bastante fácil diseñar algoritmos que generen progresiones de acordes.

3. La interfaz se implementaría mediante un lenguaje imperativo.

Según fue evolucionando el proyecto y fuimos desarrollando nuestras ideas modificamos nuestra opinión respecto a las cuestiones anteriores:

- Pasamos de pensar en un compositor automático a pensar en un asistente a la composición: Genaro tendría que ser un programa interactivo para ser más útil, liberando al compositor de tareas repetitivas (como el enlace de las voces, o la edición manual de archivos midi), pudiendo trabajar a niveles de abstracción más altos como el de las progresiones de acordes o los patrones rítmicos. También pensando en el compositor amateur, que no sabe demasiado de armonía y que podría aprender a través de Genaro, o simplemente en un músico que tiene un encargada la composición de una pieza de música electrónica y quiere plasmar rápidamente sus ideas, o recibir inspiración o ideas nuevas de Genaro, ideas que luego puede desarrollar de forma coherente utilizando su experiencia humana.

- Abandonar Sicstus por SWI-Prolog y Haskell: en un primer momento se pensó en utilizar Sicstus Prolog porque se pretedía abordar la generación de acordes como la formalización de una serie de restricciones sobre las progresiones, basadas en las reglas de la armonía, y que luego Prolog se ocupara de todo con sus búsquedas. Cuando empezamos a comprender como funcionaban las restricciones nos dimos cuenta de que era más fácil y más natural, también por las propias reglas de la armonía, diseñar el generador de acordes como un programa que mutara progresiones.
  Se continuó usando Prolog por inercia y también por su alto grado de abstracción, que facilitaba centrarse en los algoritmos y pasar más fácilmente el trámite de la implementación. Y llegó el momento de generar los archivos midi, el producto final de Genaro, para las progresiones de acordes primigenias, enlazando las voces siempre igual y aplicando siempre el mismo tipo de ritmo (el concepto de patrón rítmico aún no estaba desarrollado). Para ello se estudió el estandar midi y se definieron predicados que formalizaran una representación intermedia de la música, entre las progresiones y el midi. Se llegaron a definir dos representaciones de este tipo. La idea era que Prolog escribiera un archivo de texto con el formato de una de esas representaciones intermedias y que C++ leyera ese archivo y generara el midi correspondiente. Tras muchos problemas con este enfoque se descubrió la librería Haskore de Haskell que tenía una tipo que representaba musicas y sobre todo funciones que convertián elementos de ese tipo en archivos midi.
  Durante un tiempo Haskell se empleó solamente para el paso a midi, para lo que se definieron predicados para una representación en Prolog similar a la del Haskore. El formato de las progresiones ya estaba establecido y es el que ha perdurado hasta ahora, a partir de terminos con ese formato se producián términos en formato Prolog-Haskore con la primera versión del traductor de cifrados hecho en Prolog. Luego Haskell parseaba un fichero en ese formato y producía el midi. El módulo de los patrones rítmicos fue el primero que desarrolló en Haskell desde un principio. La facilidad de uso, potencia y flexibilidad de Haskore hicieron que el módulo traductor de cifrados se migrara a Haskell, y que todos los módulo desarrollados después (el de melodía, el de bajo y el de batería) se desarrollaran en Haskell desde un principio. También se desarrolló en Haskell el armonizador, a través del cuál se articula la otra vía de generación de progresiones de acordes.
  Mientras tanto en el módulo de Prolog hubo cambios significativos. El hecho de que Genaro se enfocara ahora como una herramiento interactiva en vez de como un compositor independiente

implicaba que la duración de las progresiones se especificara de forma exacta, no aproximada como en las primeras versiones. Debido a este se modificó la primera versión de la generación de acordes, basada en la vuelta atrás, en la que las mutaciones alargaban la progresión hasta lograr una longitud cercana a la especificada. A partir de esta versión antigua se llegó a la actual, en la que se fija exactamente la duración de la progresión y luego se la muta respetando esta duración. Este nuevo modelo de generación de progresiones también facilitó que se respetara el ritmo armónico. Lo difícil en cuanto a algoritmía de generación de música en este proyecto fue empezar, porque una vez definida la abstracción del ritmo y el espíritu mutador de Genaro, y establecida la herramienta de trabajo que es Haskore todo fué más sencillo:

- la melodía recicla el concepto de patrón rítmico pasándolo de dos dimensiones a una, y el concepto de mutaciones del generador de acordes (divide notas, junta notas), solo añade el concepto de curva melódica que es un clásico en los libros de armonía y teoría musical, no es nada nuevo.
- el bajo comparte muchas características con la melodía, y sus algoritmos aprovechan muchas ideas de los de ésta.
- La batería es simplemente un acorde que siempre es el mismo (todas las piezas de la batería sonando a la vez), al que se le aplica un patrón rítmico.

En esta última fase del desarrollo de los algoritmos de genaración música hubo una cierta evolución de las ideas iniciales en los casos de la melodía y del bajo:

- la melodía: En un principio con la primera versión de la melodía se devolvía la lista de acentos sin emplear. Esta se usaba como entrada para las funciones de mutación. Pero después de un tiempo nos dimos cuenta de que esa información estaba implícita en los silencios que forman parte de la melodía, así que se modificaron las funciones de mutación para que trabajaran con listas de música, algo mucho más sencillo y transparente.
- el bajo: Desde un principio se pensó en hacer un bajo que implementara la técnica del *walking*, técnica de composición de bajos del Jazz, que consiste en hacer que el bajo toque casi siempre notas de una misma duración fijada, sin parar, y que vaya movindose por la escala dando saltos pequeños, 'persiguiendo' a los acordes. Esa era la idea pero en un primer momento se hizo una versión tonta del bajo que para cada acorde siempre daba la misma nota, para que C++ tuviera una función a la que llamar y no huniera que integrar a última hora. Esa función debía sobreescribirse con la definitiva, pero este bajista tonto gustó a algunos desarrolladores y una versión suya modificada acabó siendo el bajista Fundamentalista. El bajista Aphex surgió como un experimento de generación que en parte fracasó, porque fué un intento de hacer el bajista Walking sin trabajar demasiado. Pero como sonaba bonito se mantuvo y acabó siendo un bajista bastate melódico, en ocasiones incluso más que la propia melodía. Y finalmente se desarrolló el bajista Walking que sí que es más o menos lo que se pensó en un principio.

- Interfaz: Se comenzó haciendo en Java pero pronto se migró a Borland C++ por la facilidad de uso y potencia de sus librerías. En el caso del interfaz y del bucle principal lo difícil vino al final del proyecto. La decisión de hacer Genaro más interactivo supuso ampliar muchísimo el interfaz, que se llenó de opciones, botones y rejillas, editores de patrones rítmicos y de curvas melódicas...A lo que hay que sumar el trabajo de coordinar las llamadas entre lenguajes y a los programas externos,

así como diseñar la estructura de los proyectos de Genaro, que acabaron complicándose bastante. Así que pasamos de un interfaz casi simbólico a un interfaz que ofrece muchísimas posibilidades de manipulación haciendo Genaro mucho más útil y divertido.

Resumiendo, se pasó de pensar en un compositor autónomo teniendo solamente claro unos esbozos de cómo generar progresiones, a una herramienta interactiva que sirve como asistente a la composición, y que genera acompañamientos, melodías y bajos, para cualquier número de pistas y para una gran variedad de fuentes de sonido.

Bibliografía

[1] Enric Herrera. *Teoría Musical y Armonía Moderna, Vol 1 y 2*. Antoni Bosch, editor SA, 13 edition, 2004.

[2] Dave Benson. *Mathematics and Music*. University of Georgia, `http://www.math.uga.edu/~djb/html/math-music.html`, 2004.

[3] Paul Hudak. *Haskore Music Tutorial*. Yale University, Department of Computer Science, `http://www.cs.lth.se/EDA120/assignment2/tutorial.pdf`, 2000.

[4] Tomas Bautista, Tobias Oetiker, Huber Partl, Irene Hyna y Elisabeth Schlegl. *Una Descripción de LaTeX2ε*, Centro de Microelectrónica Aplicada de la Universidad de Las Palmas de Gran Canaria, 1996.

[5] Jeroen Fokker. *Functional Parsers*. Berlin, `http://www.cs.uu.nl/people/jeroen/article/parsers/index.html`, 1995

[6] Han-Wen Nienhuys, Jan Nieuwenhuizen, Jurgen Reuter y Rune Zedeler. *GNU LilyPond*. `http://www.lilypond.org/doc/v2.0/ Documentation/user/out-www/lilypond.pdf`, 2003.

[7] Teresa Hortalá González y Javier Leach Albert. *Programación Lógica*. Universidad Complutense de Madrid, `http://www.fdi.ucm.es/profesor/leach/PL/proglog.pdf`, 2004.

[8] Mario Rodríguez Artalejo. *Programación Funcional*. Universidad Complutense de Madrid, `http://www.fdi.ucm.es/profesor/mario/PF/cursoPF.pdf`, 2004

[9] Miloslav Nic. *Haskell reference*. `http://zvon.org/other/haskell/ Outputglobal/index.html`

Los abajo firmantes, expresan su conformidad a autorizar a la Universidad Complutense de Madrid, a difundr y utilizar con fines académicos, en ningún caso comerciales, tanto la presente memoria, como el código del proyecto, el prototipo desarrollado y la documentación que se considere necesaria.

Fdo. Javier Gómez Santos

Fdo. Juan Rodríguez Hortalá

Fdo. Roberto Torres de Alba